3em

Understanding Deep Learning

Simon J.D. Prince

October 1, 2022

I would really appreciate help improving this document. No detail too small! Please mail suggestions, factual inaccuracies, ambiguities, questions, and errata to udlbookmail@gmail.com.

Contents

Chapter 1

Introduction

Chapter 2

Supervised learning

A *supervised learning model* defines a mapping from one or more inputs to one or more outputs. For example, the input might be the age and mileage of a secondhand Toyota Prius and the output might be the estimated value of the car in dollars.

The model is just a mathematical equation; when the inputs are passed through this equation, it computes the prediction, and this is termed *inference*. This model equation also contains *parameters*. Different parameter values change the outcome of the computation; the model equation describes a family of possible relationships between inputs and outputs, and the parameters specify the particular relationship.

When we *train* or *learn* a model, we choose the parameters so that the relationship between inputs and outputs is correct. The learning algorithm takes a training set of input/output pairs and manipulates the parameters until the inputs predict their corresponding outputs as closely as possible. If the model works well for these training pairs, then we hope that it will make good predictions for new inputs where the true output is unknown.

The goal of this chapter is to expand on these ideas. First, we'll describe this framework more formally and introduce some notation. Then we'll work through a simple example in which we use a straight line to describe the relationship between input and output. This linear model is both familiar and easy to visualize, but nevertheless illustrates all the main ideas of supervised learning.

2.1 Supervised learning overview

In supervised learning, we aim to build a model that takes input data $\mathbf{x}$ and outputs a prediction $\mathbf{y}$. For simplicity, we'll assume that both the input $\mathbf{x}$ and output $\mathbf{y}$ are vectors of a predetermined and fixed size. Moreover, the elements of each vector are always organized in the same way, so in the Prius example above, the input vector $\mathbf{x}$ would always contain the age of the car and then the mileage, in that order. This is known as *structured* or *tabular* data.

Notation
Appendix A

To make the prediction, we need a model $\mathbf{f}[\bullet]$ that takes input $\mathbf{x}$ and returns $\mathbf{y}$:

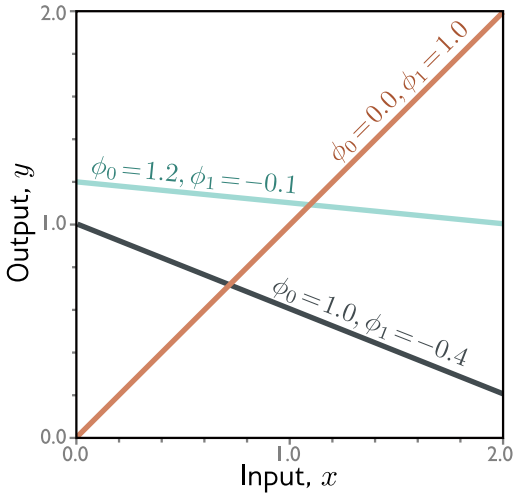

Figure 2.1 Linear regression model. For a given choice of parameters $\boldsymbol{\phi} = [\phi_0, \phi_1]^T$, the model makes a prediction for the output (y-axis) based on the input (x-axis). Different choices for the y-intercept ϕ_0 and the slope ϕ_1 change these predictions (cyan, orange, and gray lines). The linear regression model (equation 2.4) defines a family of input/output relations (lines) and the parameters determine the member of the family (the particular line).

$$\mathbf{y} = \mathbf{f}[\mathbf{x}]. \tag{2.1}$$

When we compute the prediction $\mathbf{y}$ from the input $\mathbf{x}$, we call this *inference*.

The model is just a mathematical equation with a fixed form. It represents a family of different relations between the input and the output. The model also contains *parameters* $\boldsymbol{\phi}$. The choice of parameters determines the particular relation between input and output, and so really, we should write:

$$\mathbf{y} = \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]. \tag{2.2}$$

When we talk about *learning* or *training* a model, we mean that we attempt to find parameters $\boldsymbol{\phi}$ that make sensible output predictions from the input. We learn these parameters using a *training dataset* of I pairs of input and output examples $\{\mathbf{x}_i, \mathbf{y}_i\}$. We aim to select parameters that map each training input to its associated output as closely as possible. We quantify the degree of mismatch in this mapping with the *loss* $\mathrm{L}[\boldsymbol{\phi}]$. This is a scalar value that summarizes how poorly the model predicts the training outputs from their corresponding inputs for parameters $\boldsymbol{\phi}$. So when we train the model, we are seeking parameters $\hat{\boldsymbol{\phi}}$ that minimize the loss:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmin}} \left[\mathrm{L}\left[\boldsymbol{\phi}\right]\right]. \tag{2.3}$$

If the final loss is small, then we have found model parameters that accurately predict the training outputs $\mathbf{y}_i$ from the training inputs $\mathbf{x}_i$.

After training the model, we must now assess its performance; we run the model on separate *test data* to see how well it *generalizes* to examples that it did not have access to during training. If the test performance is adequate, then we are ready to apply the model to new data.

2.2 Linear regression example

Let's now make these ideas concrete with a simple example. We'll consider a model $y = \mathrm{f}[x, \boldsymbol{\phi}]$ that predicts a single output y from a single input x. Then we'll develop a loss function, and finally, we'll discuss model training.

2.2.1 1D linear regression model

A *1D linear regression model* describes the relationship between input x and output y as a straight line:

$$\begin{aligned} y &= \mathrm{f}[x, \boldsymbol{\phi}] \\ &= \phi_0 + \phi_1 x. \end{aligned} \tag{2.4}$$

This model has two parameters $\boldsymbol{\phi} = [\phi_0, \phi_1]^T$, where ϕ_0 is the y-intercept of the line and the ϕ_1 is the slope. Different choices for the intercept and slope result in different relations between input and output (figure 2.1). Hence, equation 2.4 defines a family of possible input-output relations (all possible lines), and the choice of parameters determines the member of this family (the particular line).

2.2.2 Loss

For this model, the training dataset (figure 2.2a) consists of I input/output pairs $\{x_i, y_i\}$. Figures 2.2b-d show three lines defined by three sets of parameters. The green line in figure 2.2d describes the data more accurately than the other two since it is much closer to the data points. However, we need a principled approach for deciding which parameters $\boldsymbol{\phi}$ are better than others. The goal will be to associate a numerical value with each choice of parameters which we refer to as the *loss*. This quantifies the degree of mismatch between the model and the data; a lower loss means a better fit.

The mismatch is captured by the deviation between the model predictions $\mathrm{f}[x_i, \boldsymbol{\phi}]$ (height of the line at x_i) and the ground truth output values y_i. These deviations are depicted as orange dashed lines in figures 2.2b-d. We quantify the total mismatch, *training error*, or *loss* as the sum of the squares of these deviations for all I training pairs:

$$\begin{aligned} L[\phi] &= \sum_{i=1}^{I} \left(\mathrm{f}[x_i, \boldsymbol{\phi}] - y_i\right)^2 \\ &= \sum_{i=1}^{I} \left(\phi_0 + \phi_1 x_i - y_i\right)^2. \end{aligned} \tag{2.5}$$

Since the best parameters minimize this expression, we call this a *least-squares* loss. The squaring operation means that the direction of the deviation (i.e., whether the line is

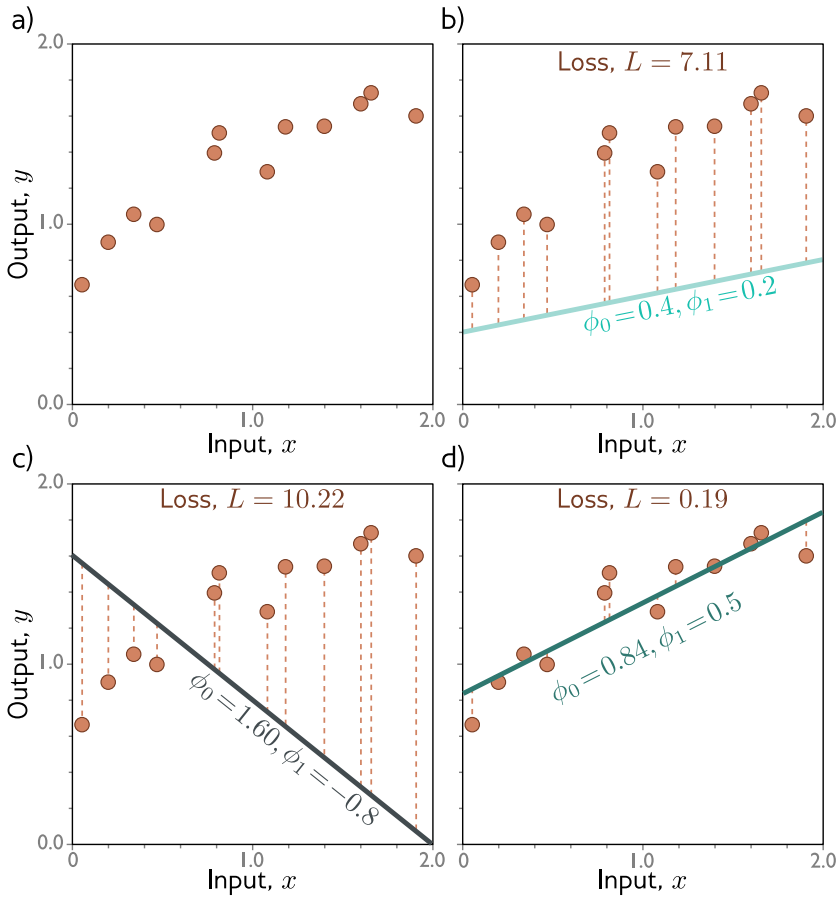

Figure 2.2 Linear regression training data, model, and loss. a) The training data (orange points) consist of $I = 12$ input/output pairs $\{x_i, y_i\}$. b-d) Each panel shows the linear regression model with different parameters. Depending on the choice of slope and intercept parameters $\boldsymbol{\phi} = [\phi_0, \phi_1]^T$, the model errors (orange dashed lines) may be larger or smaller. The loss function $L[\boldsymbol{\phi}]$ is the sum of the squares of these errors. The parameters that define the lines in panels (b) and (c) have large losses $L = 7.11$ and $L = 10.22$ respectively because the models fit badly. The loss $L = 0.19$ in panel (d) is much smaller because the model fits well; in fact, this has the smallest loss of all possible lines and so these are the best possible parameters.

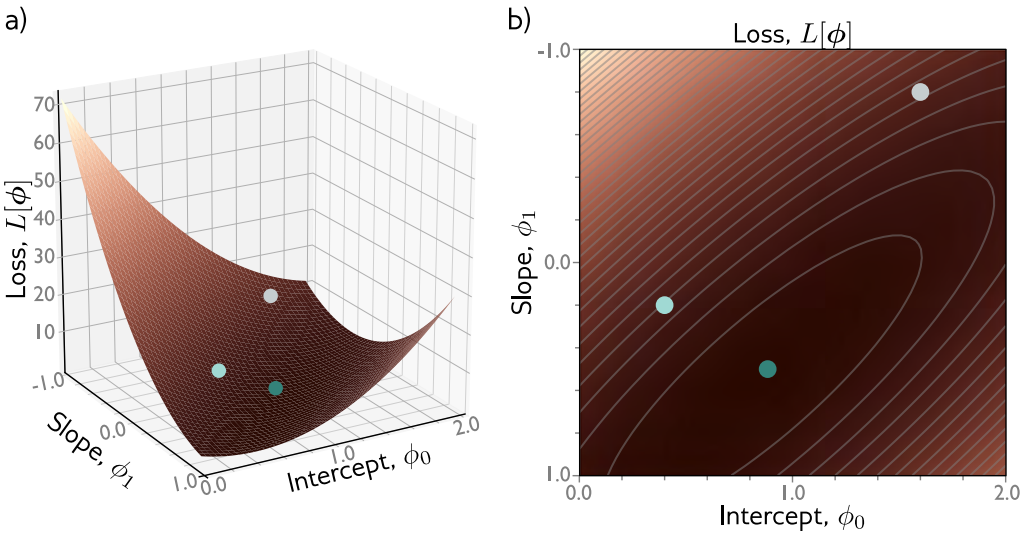

Figure 2.3 Loss function for linear regression model with the dataset in figure 2.2a. a) Each combination of parameters $\boldsymbol{\phi} = [\phi_0,\phi_1]$ has an associated loss. The resulting loss function $L[\boldsymbol{\phi}]$ can be visualized as a surface. The three circles represent the three lines from figure 2.2b-d. b) The loss can also be visualized as a heatmap, where brighter regions represent larger losses; here we are looking straight down at the surface in (a) from above. The best fitting line (figure 2.2d) has the parameters with the smallest loss (green circle).

above or below the data) is unimportant. There are also theoretical reasons for this choice which we return to in chapter 5.

The loss L is a function of the parameters $\boldsymbol{\phi}$; it will be larger when the model fit is poor (figure 2.2b,c) and smaller when it is good (figure 2.2d). Considered in this light, we term $L[\boldsymbol{\phi}]$ the *loss function* or *cost function*. The goal is to find the parameters $\hat{\boldsymbol{\phi}}$ that minimize this quantity:

$$\begin{aligned}\hat{\boldsymbol{\phi}} &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[L[\boldsymbol{\phi}]\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[\sum_{i=1}^{I}\left(\mathrm{f}[x_i,\boldsymbol{\phi}]-y_i\right)^2\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[\sum_{i=1}^{I}\left(\phi_0+\phi_1 x_i-y_i\right)^2\right]. \qquad (2.6)\end{aligned}$$

There are only two parameters (the intercept ϕ_0 and slope ϕ_1) and so we can calculate the loss for every combination of values and visualize the loss function as a surface (figure 2.3). The "best" parameters are at the minimum of this surface.

Problems 2.1-2.2

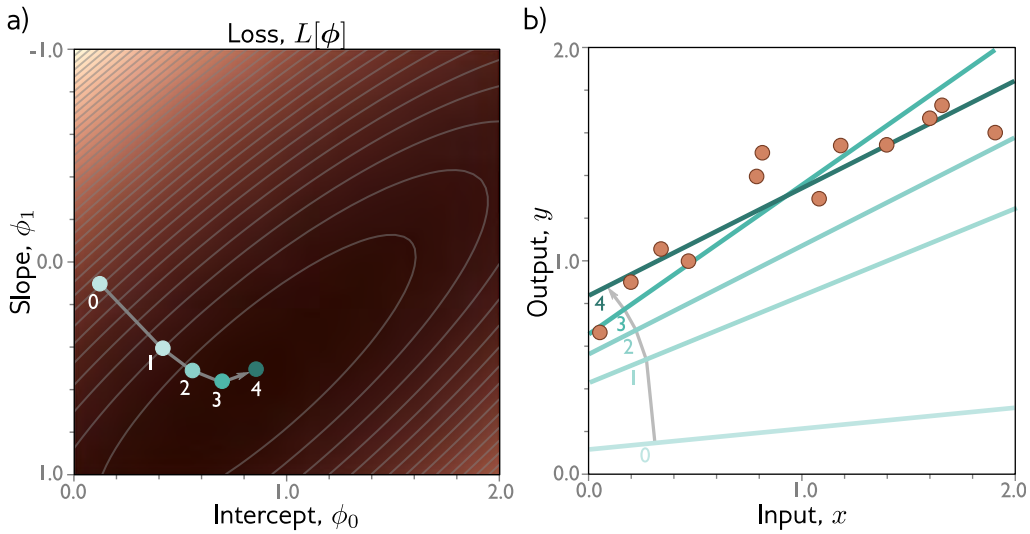

Figure 2.4 Training the linear regression model. The goal is to find the slope/intercept parameters that correspond to the smallest loss. a) Iterative training algorithms initialize the parameters randomly and then improve them by "walking downhill" until no further improvement can be made. Here we start at position 0 and move a certain distance downhill (perpendicular to the contours) to position 1. Then we re-calculate the downhill direction and move to position 2. Eventually, we wind up at the minimum of the function (position 4). b) Each position 0-4 from panel (a) corresponds to a different y-intercept and slope and hence represents a different line. As the loss decreases, these lines fit the data more closely.

2.2.3 Training

The process of finding parameters that minimize the loss is termed *model fitting*, *training*, or *learning*. The basic method is to choose the initial parameters randomly and then improve them by "walking down" the loss function until we reach the bottom (figure 2.4). One way to do this is to measure the gradient of the surface at the current position and take a step in the direction that is most steeply downhill. Then we repeat this process until the gradient is flat and we can improve no further.[1]

[1]In fact, this iterative approach is not necessary for the linear regression model. Here, it's possible to find closed form expressions for the parameters. However, this *gradient descent* approach works in more complex models, where there is no closed from solution, and where there are too many parameters to evaluate the loss for every combination of values.

2.2.4 Testing

Having trained the model, we want to know how it will perform in the real world. We do this by computing the loss on a separate set of *test data*. The degree to which the prediction accuracy *generalizes* to the test data depends in part on how representative and complete the training data was. However, it also depends on the model complexity. A simple model might not be able to capture the true relationship between input and output. Conversely, a complex model may describe statistical peculiarities of the training data that are atypical and lead to unusual predictions. This is known as *overfitting*.

2.3 Summary

A supervised learning model is a function $\mathbf{y} = \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ that relates inputs $\mathbf{x}$ to outputs $\mathbf{y}$. The particular relationship is determined by parameters $\boldsymbol{\phi}$. To train the model, we define a loss function $L[\boldsymbol{\phi}]$ over a training dataset $\{\mathbf{x}_i, \mathbf{y}_i\}$. This quantifies the mismatch between the model predictions $\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ and observed outputs $\mathbf{y}_i$ as a function of the parameters $\boldsymbol{\phi}$. Then we search for the parameters that minimize the loss. We evaluate the model on a different set of test data to see how well it generalizes to new inputs.

In chapters 3-5, we expand on these ideas. First, we tackle the model itself; linear regression has the obvious drawback that it can only describe the relation between the input and output as a straight line. To resolve this problem, we introduce shallow neural networks in chapter 3. These are only slightly more complex than linear regression but describe a much larger family of relationships between the input and output. In chapter 4 we introduce deep neural networks, which are just as expressive, but can describe complex functions with fewer parameters and work better in practice.

In chapter 5, we investigate loss functions for different tasks and reveal the theoretical underpinnings of the least-squares loss function. In chapters 6 and 7, we discuss the training process and introduce the famous backpropagation algorithm. In chapter 8 we discuss how to measure model performance. In chapter 9, we consider *regularization* techniques, which aim to improve that performance.

Notes

Loss functions vs. cost functions: In much of machine learning and in this book, the terms loss function and cost function are used interchangeably. However, more properly a loss function is the individual term associated with a data point (i.e., each of the squared terms on the right-hand side of equation 2.5) and the cost function is the overall quantity that is minimized (i.e., the entire right-hand side of equation 2.5). A cost function can contain additional terms that are not associated with individual data points. More generally, an *objective function* is any function that is to be maximized or minimized.

Generative vs. discriminative models: The models $\mathbf{y} = \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ in this chapter are *discriminative models*. These make an output prediction $\mathbf{y}$ from real-world measurements $\mathbf{x}$. Another approach is to build a *generative model* $\mathbf{x} = \mathbf{g}[\mathbf{y}, \boldsymbol{\phi}]$, in which the real-world measurements $\mathbf{x}$ are computed as a function of the output $\mathbf{y}$.

Problem 2.3

The generative approach has the disadvantage that it doesn't directly predict $\mathbf{y}$. To perform inference, we must invert the generative equation as $\mathbf{y} = \mathbf{g}^{-1}[\mathbf{x}, \boldsymbol{\phi}]$ and this may be difficult. However, generative models have the advantage that we can build in prior knowledge about how the data were generated. For example, if we wanted to predict the 3D position and orientation $\mathbf{y}$ of a car in an image $\mathbf{x}$, then we could build knowledge about car shape, 3D geometry, and light transport into the function $\mathbf{x} = \mathbf{g}[\mathbf{y}, \boldsymbol{\phi}]$.

This seems like a good idea, but in fact, discriminative models dominate modern machine learning; the advantage gained from exploiting prior knowledge in generative models is usually trumped by brute force learning of very flexible discriminative models from large amounts of training data.

Problems

Problem 2.1 To walk "downhill" on the loss function (equation 2.5), we measure its slope with respect to the parameters ϕ_0 and ϕ_1. Calculate expressions for the slopes $\partial L/\partial\phi_0$ and $\partial L/\partial\phi_1$.

Problem 2.2 Show that we can find the minimum of the loss function in closed form by setting the expression for the derivatives from problem 2.1 to zero and solving for ϕ_0 and ϕ_1. Note that this works for linear regression but not for more complex models; this is why we use iterative model fitting methods like gradient descent (figure 2.4).

Problem 2.3 Consider reformulating linear regression as a generative model so we have $x = \mathrm{g}[y, \boldsymbol{\phi}] = \phi_0 + \phi_1 y$. What is the new loss function? Find an expression for the inverse function $y = \mathrm{g}^{-1}[x, \boldsymbol{\phi}]$ that we would use to perform inference. Will this model make the same predictions as the discriminative version for a given training dataset $\{x_i, y_i\}$?

Chapter 3

Shallow neural networks

Chapter 2 introduced supervised learning using 1D linear regression. However, this model can only describe the input/output relationship as a line. This chapter introduces shallow neural networks. These describe piecewise linear functions and are flexible enough to describe arbitrarily complex relationships between multi-dimensional inputs and outputs.

3.1 Neural network example

In their general form, shallow neural networks are functions $\mathbf{y} = \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ with parameters $\boldsymbol{\phi}$ that map multivariate inputs $\mathbf{x}$ to multivariate outputs $\mathbf{y}$. We'll defer a full definition until section 3.4 and introduce the main ideas using a simple example in which we map a scalar input x to a scalar output y:

$$\begin{aligned} y &= \mathrm{f}[x, \boldsymbol{\phi}] \\ &= \phi_0 + \phi_1 \mathrm{a}[\theta_{10} + \theta_{11} x] + \phi_2 \mathrm{a}[\theta_{20} + \theta_{21} x] + \phi_3 \mathrm{a}[\theta_{30} + \theta_{31} x]. \end{aligned} \tag{3.1}$$

This model has ten parameters $\boldsymbol{\phi} = \{\phi_0, \phi_1, \phi_2, \phi_3, \theta_{10}, \theta_{11}, \theta_{20}, \theta_{21}, \theta_{30}, \theta_{31}\}$.

The calculation in equation 3.1 can be broken down into three parts: first, we compute three linear functions of the input data ($\theta_{10} + \theta_{11} x$, $\theta_{20} + \theta_{21} x$, and $\theta_{30} + \theta_{31} x$). Second, we pass the three results through an *activation function* $\mathrm{a}[\bullet]$. Finally, we weight the three resulting activations with ϕ_1, ϕ_2, and ϕ_3, sum them, and add an offset ϕ_0.

To complete the description, we must define the activation function $\mathrm{a}[\bullet]$. There are several possibilities, but the most common is the *rectified linear unit* or *ReLU*:

Problem 3.1

$$\mathrm{a}[z] = \mathrm{ReLU}[z] = \begin{cases} 0 & z < 0 \\ z & z \geq 0 \end{cases}. \tag{3.2}$$

This returns the input when it is positive and zero otherwise (figure 3.1).

It's probably not obvious which family of input/output relations is represented by equation 3.1. Nonetheless, the ideas from the previous chapter are all applicable. Equation 3.1 represents a family of functions, where the particular member of the family

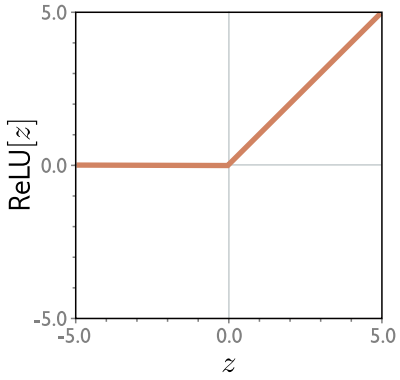

Figure 3.1 Rectified linear unit (ReLU). This activation function returns zero if the input is less than zero and returns the input unchanged if it is greater than zero. In other words, it clips negative values to zero. Note that there are many other possible choices for the activation function (see figure 3.13) but the ReLU is the most used and easiest to understand.

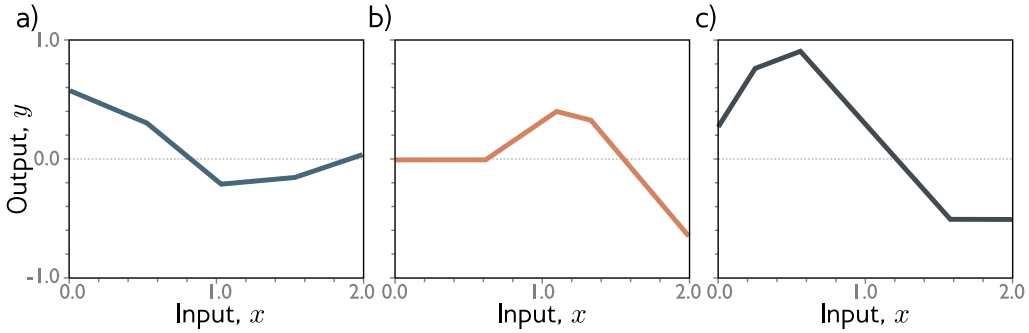

Figure 3.2 Family of functions defined by equation 3.1. a–c) Functions for three different choices of the ten parameters $\boldsymbol{\phi}$. In each case, the input/output relation is piecewise linear. However, the positions of the joints, the slopes of the linear regions between them, and the overall height vary.

depends on the ten parameters in $\boldsymbol{\phi}$. If we know these parameters, we can perform inference (predict y) by evaluating the equation for a given input x. Given a training dataset $\{x_i, y_i\}_{i=1}^{I}$, we can define a least squares loss function $\mathrm{L}[\boldsymbol{\phi}]$, and so measure how effectively the model describes this dataset for any given parameter values $\boldsymbol{\phi}$. To train the model, we search for the values $\hat{\boldsymbol{\phi}}$ that minimize this loss.

3.1.1 Neural network intuition

In fact, equation 3.1, represents a family of continuous piecewise linear functions (figure 3.2) with up to four linear regions. We'll now break down equation 3.1 and show *why* it describes this family. To make this easier to understand, we'll split the function into two parts. First, we introduce the intermediate quantities:

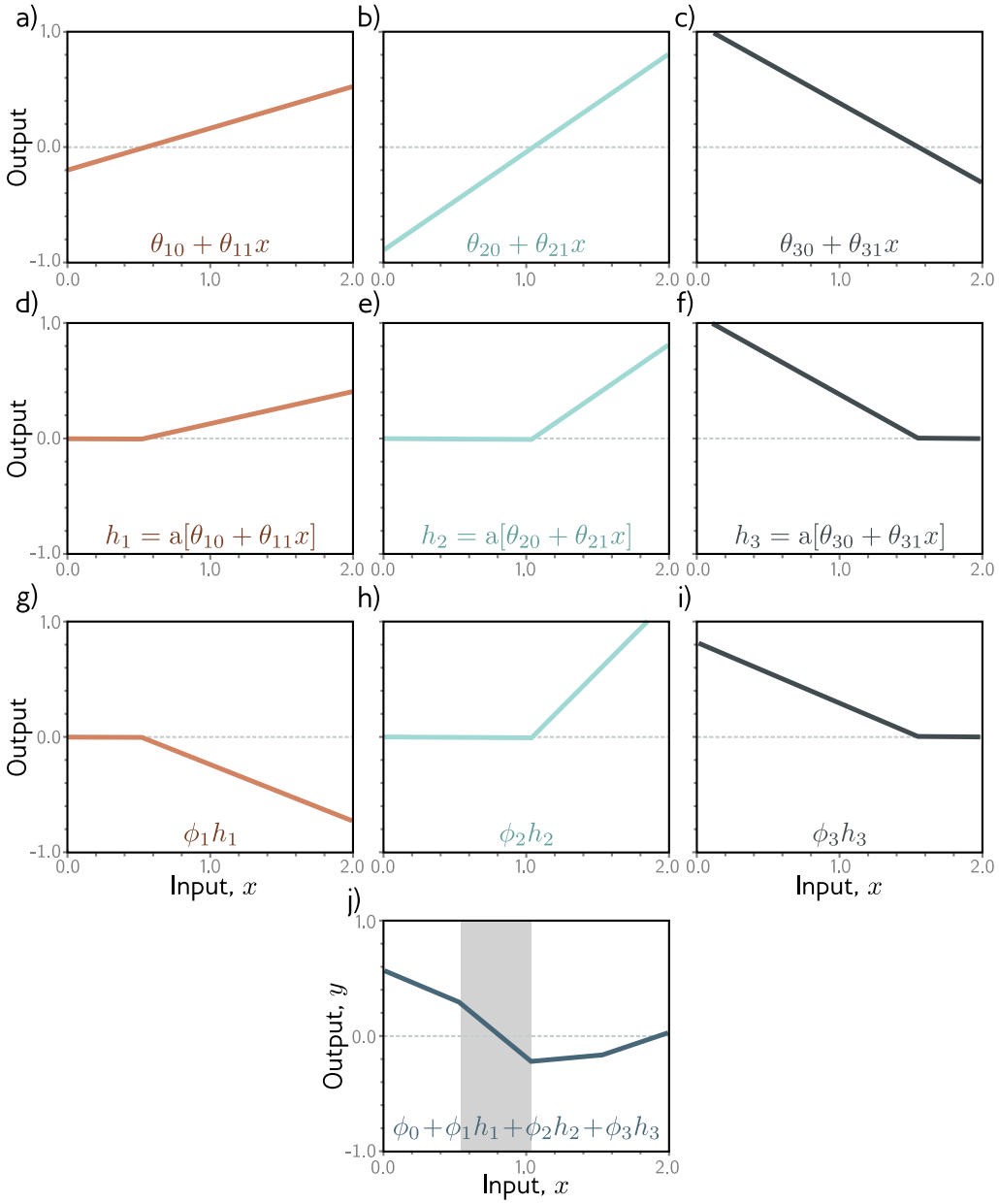

Figure 3.3 Computation for function in figure 3.2a. a–c) The input x is passed through three linear functions, each with a different y-intercept $\theta_{\bullet 0}$ and slope $\theta_{\bullet 1}$. d–f) Each line is passed through the ReLU activation function, which clips negative values to zero. g–i) The three clipped lines are then weighted (scaled) by the parameters ϕ_1, ϕ_2, and ϕ_3, respectively. h) Finally, the clipped and weighted functions are summed and an offset ϕ_0 that controls the height is added. Each of the four linear regions corresponds to a different activation pattern in the hidden units. In the shaded region, h_2 is inactive (clipped) but h_1 and h_3 are both active.

$$\begin{aligned} h_1 &= \mathrm{a}[\theta_{10} + \theta_{11}x] \\ h_2 &= \mathrm{a}[\theta_{20} + \theta_{21}x] \\ h_3 &= \mathrm{a}[\theta_{30} + \theta_{31}x], \end{aligned} \tag{3.3}$$

where we refer to h_1, h_2, and h_3 as *hidden units*. Second, we compute the output by combining these hidden units with a linear function:[1]

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3. \tag{3.4}$$

Figure 3.3 shows the flow of computation that creates the function in figure 3.2a. Each hidden unit contains a linear function $\theta_{\bullet 0} + \theta_{\bullet 1}x$ of the input, and that line is clipped by the ReLU function $a[\bullet]$ below zero. The positions where the three lines cross zero become the three "joints" in the final output. The three clipped lines are then weighted by ϕ_1, ϕ_2, and ϕ_3 respectively. Finally, the offset ϕ_0 is added, which controls the overall height of the final function.

Problems 3.2-3.8

Each linear region in figure 3.3j corresponds to a different *activation pattern* in the hidden units. When a unit is clipped, we refer to it as *inactive*, and when it is not clipped, we refer to it as *active*. For example, the shaded region receives contributions from h_1 and h_3 (which are active) but not from h_2 (which is inactive). The slope of each linear region is determined by (i) the original slopes $\theta_{\bullet 1}$ of the active inputs for this region, and (ii) the weights $\phi_\bullet$ that were subsequently applied. For example, the slope in the shaded region is $\theta_{11}\phi_1 + \theta_{31}\phi_3$.

Each hidden unit contributes one "joint" to the function, so with three hidden units, there can be four linear regions. However, only three of the slopes of these regions are independent; the fourth is either zero (if all the hidden units are inactive in this region) or is a sum of slopes from the other regions.

Problem 3.9

3.1.2 Depicting neural networks

We have been discussing a neural network with one input, one output, and three hidden units. We visualize this network in figure 3.4a. The input is on the left, the hidden units are in the middle, and the output is on the right. Viewed in this way, each connection represents one of the ten parameters. To simplify this representation, we do not typically draw the intercept parameters, and so this network would usually be depicted as in figure 3.4b.

[1] For the purposes of this book, a linear function has the form $z' = \phi_0 + \sum_i \phi_i z_i$. Any other type of function is nonlinear. For instance, the ReLU function (equation 3.2) and the example neural network that contains it (equation 3.1) are both nonlinear. See notes at end of chapter for more clarification.

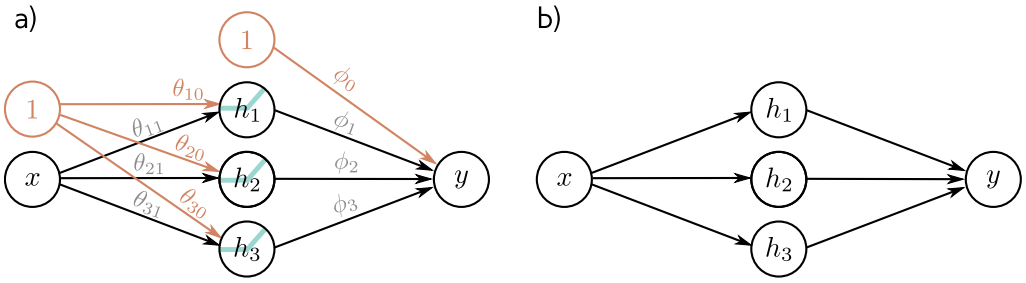

Figure 3.4 Depicting neural networks. a) The input x is on the left and the output y on the right. The three hidden units are in the center. The computation flows from left to right. The input is used to compute the values of the three hidden units and these values are combined to create the output. Each of the ten arrows represents one of the parameters. The intercepts are in orange and the slopes in black. Each parameter multiplies its source variable and adds the result to its target variable. For example, we multiply the parameter ϕ_1 by its source h_1 and add it to y. We introduce additional nodes containing ones (orange circles) to incorporate the offsets. The ReLU functions are applied in the hidden units. b) More typically, the intercepts, ReLU functions, and parameter names are omitted, and so this simpler depiction would be used to represent the same network.

3.2 Universal approximation theorem

In the previous section, we introduced an example neural network with one input, one output, ReLU activation functions, and three hidden units. Let's now generalize this slightly and consider the case with D hidden units where the d^{th} hidden unit is:

$$h_d = \mathrm{a}[\theta_{d0} + \theta_{d1}x], \tag{3.5}$$

and these are combined linearly to create the output:

$$y = \phi_0 + \sum_{d=1}^{D} \phi_d h_d. \tag{3.6}$$

The number of hidden units in a shallow network is a measure of the *network capacity*. With ReLU activation functions, the output of a network with D hidden units is a piecewise linear function with at most $D+1$ linear regions. As we increase the number of hidden units, the model can approximate more complex functions.

Problem 3.10

Indeed, with enough capacity (hidden units), a shallow network can describe any continuous 1D function defined on a compact subset of the real line to arbitrary precision. To see this, consider that every time we add a hidden unit, we add another linear region to the function. As these linear regions become more numerous, they represent smaller and smaller sections of the function, which are increasingly well approximated by a line (figure 3.5). The ability of a neural network to approximate any continuous function can be formally proven and this is known as the *universal approximation theorem*.

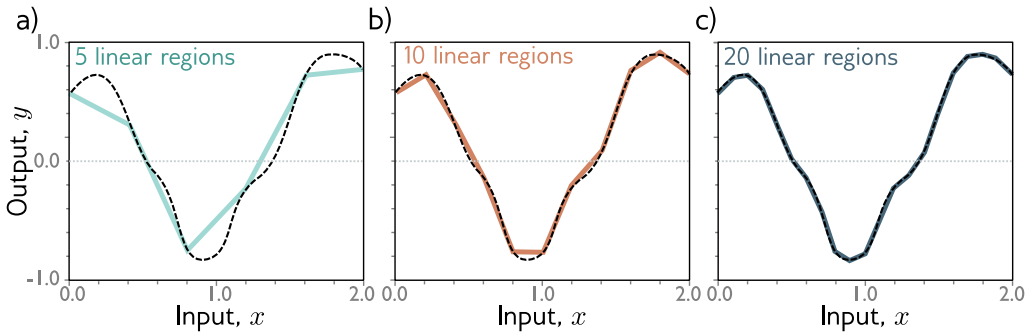

Figure 3.5 Approximation of 1D function (dashed line) by piecewise linear model. a–c) As the number of regions increases, the model becomes closer and closer to the continuous function. A neural network with a scalar input creates one extra piecewise linear region per hidden unit. The universal approximation theorem provides a formal proof that with enough hidden units a shallow neural network can describe any continuous function defined on a compact subset of $\mathbb{R}^D$ to arbitrary precision.

3.3 Multivariate inputs and outputs

In the above example, the network has a single scalar input x and a single scalar output y. However, the universal approximation theorem also holds for the more general case where the network maps multivariate inputs $\mathbf{x} = [x_1, x_2, \ldots x_{D_i}]^T$ to multivariate output predictions $\mathbf{y} = [y_1, y_2, \ldots y_{D_o}]^T$. In this section, we explore how the model can be extended to predict multivariate outputs. Then we'll consider multivariate inputs. Finally, in section 3.4 we'll present a general definition of a shallow neural network.

3.3.1 Visualizing multivariate outputs

To extend the network to multivariate outputs $\mathbf{y}$, we simply use a different linear function of the hidden units for each output. So, a network with a scalar input x, four hidden units h_1, h_2, h_3, and h_4, and a 2D multivariate output $\mathbf{y} = [y_1, y_2]^T$ would be defined as:

$$\begin{aligned} h_1 &= \mathrm{a}[\theta_{10} + \theta_{11}x] \\ h_2 &= \mathrm{a}[\theta_{20} + \theta_{21}x] \\ h_3 &= \mathrm{a}[\theta_{30} + \theta_{31}x] \\ h_4 &= \mathrm{a}[\theta_{40} + \theta_{41}x], \end{aligned} \tag{3.7}$$

and

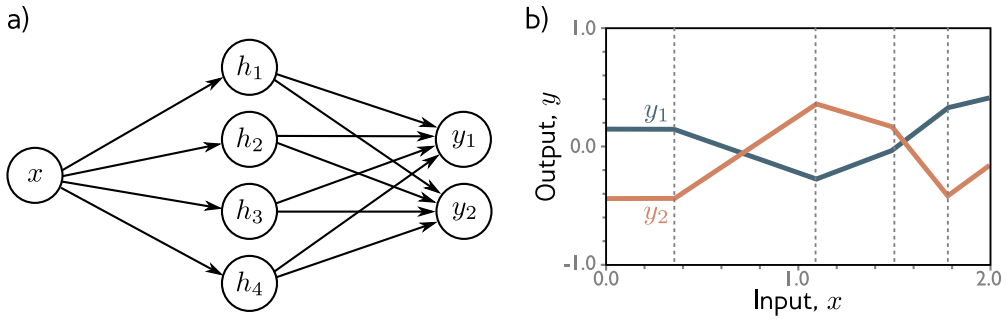

Figure 3.6 Network with one input, four hidden units, and two outputs. a) Visualization of network structure. b) This network produces two piecewise linear functions $y_1[x]$ and $y_2[x]$. The four "joints" of these functions (at vertical dotted lines) are constrained to be in the same place since they share the same hidden units, but the slopes and overall height may differ.

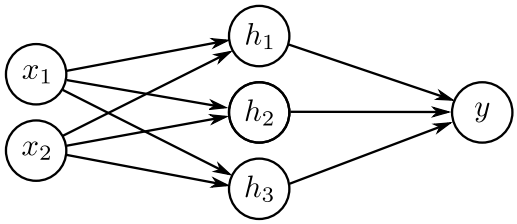

Figure 3.7 Visualization of neural network with 2D multivariate input $\mathbf{x} = [x_1, x_2]^T$ and scalar output y.

$$\begin{aligned} y_1 &= \phi_{10} + \phi_{11}h_1 + \phi_{12}h_2 + \phi_{13}h_3 + \phi_{14}h_4 \\ y_2 &= \phi_{20} + \phi_{21}h_1 + \phi_{22}h_2 + \phi_{23}h_3 + \phi_{24}h_4. \end{aligned} \tag{3.8}$$

The two outputs are two different linear functions of the hidden units.

As we saw in figure 3.3, the "joints" in the piecewise functions are determined by where the initial linear functions $\theta_{\bullet 0} + \theta_{\bullet 1}x$ are clipped by the ReLU functions $a[\bullet]$ at the hidden units. Since both outputs y_1 and y_2 are different linear functions of the same four hidden units, it follows that the four "joints" in each must be in the same place. However, the slopes of the linear regions and the overall vertical offset will differ in general (figure 3.6).

Problem 3.11

3.3.2 Visualizing multivariate inputs

To cope with multivariate inputs $\mathbf{x}$, we extend the linear relations between the input and the hidden units. So a network with two inputs $\mathbf{x} = [x_1, x_2]^T$ and a scalar output y (figure 3.7) might have three hidden units that are defined by:

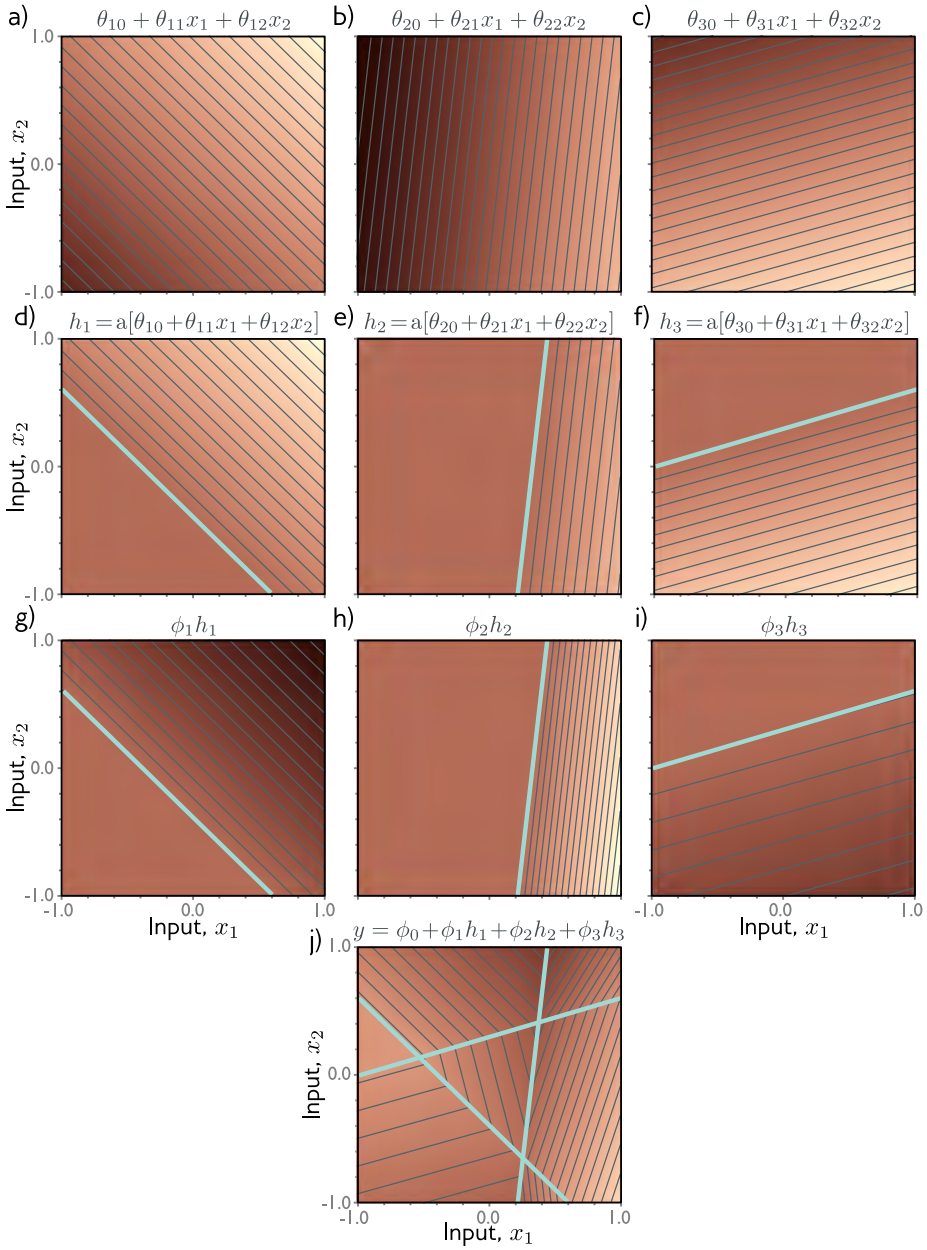

Figure 3.8 Processing in network with two inputs $\mathbf{x} = [x_1, x_2]^T$, three hidden units h_1, h_2, h_3, and one output y. a–c) The input to each hidden unit is a linear function of the two inputs, which corresponds to an oriented plane. Brightness indicates height of function, thin lines are contours. d–f) Each plane is clipped by the ReLU activation function (cyan lines are equivalent to "joints" in figures 3.3d–f). g-h) The clipped planes are then weighted, and j) summed together with an offset that determines the overall height of the surface. The result is a continuous surface made up of convex piecewise linear polygonal regions.

$$\begin{aligned} h_1 &= \mathrm{a}[\theta_{10} + \theta_{11}x_1 + \theta_{12}x_2] \\ h_2 &= \mathrm{a}[\theta_{20} + \theta_{21}x_1 + \theta_{22}x_2] \\ h_3 &= \mathrm{a}[\theta_{30} + \theta_{31}x_1 + \theta_{32}x_2], \end{aligned} \tag{3.9}$$

where there is now one slope parameter for each input. The hidden units are combined to form the output in the usual way:

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3.$$

Figure 3.8 illustrates the processing of this network. Each hidden unit receives a linear combination of the two inputs, which forms an oriented plane in the 3D input/output space. The activation function clips negative values of these planes to zero. The clipped planes are then recombined in a second linear function (equation 3.10) to create a continuous piecewise linear surface consisting of convex polygonal regions (figure 3.8j). Each region corresponds to a different activation pattern. For example, in the central triangular region, the first and third hidden units are active and the second is inactive.

Problems 3.12–3.13

Appendix C.5 convex region

When there are more than two inputs to the model, it becomes difficult to visualize. However, the interpretation is similar. The output will be a continuous piecewise linear function of the input, where the linear regions are now convex polytopes in the multidimensional input space.

Note that as the input dimensions grow, the number of linear regions increases rapidly (figure 3.9). To get a feeling for just how rapidly, consider that each hidden unit defines a hyperplane that delineates the part of space where this unit is active from the part where it is not (cyan lines in 3.8d–f). If we had the same number of hidden units as input dimensions D_i, then we could align each hyperplane with one of the coordinate axes (figure 3.10). For two input dimensions, this would divide the space into four quadrants, for three dimensions this would create eight octants, and for D_i dimensions this would create 2^{D_i} orthants. Shallow neural networks usually have a larger number of hidden units than input dimensions, so they typically create more than 2^{D_i} linear regions.

3.4 Shallow neural networks: general case

We have described several example shallow networks to help develop intuition about how they work. We now define a general equation for a shallow neural network $\mathbf{y} = \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ that maps a multidimensional input $\mathbf{x} \in \mathbb{R}^{D_i}$ to a multidimensional output $\mathbf{x} \in \mathbb{R}^{D_o}$ using $\mathbf{h} \in \mathbb{R}^{D}$ hidden units. Each hidden unit is computed as:

$$h_d = \mathrm{a}\left[\theta_{d0} + \sum_{i=1}^{D_i} \theta_{di} x_i\right], \tag{3.10}$$

and these are combined linearly to create the output:

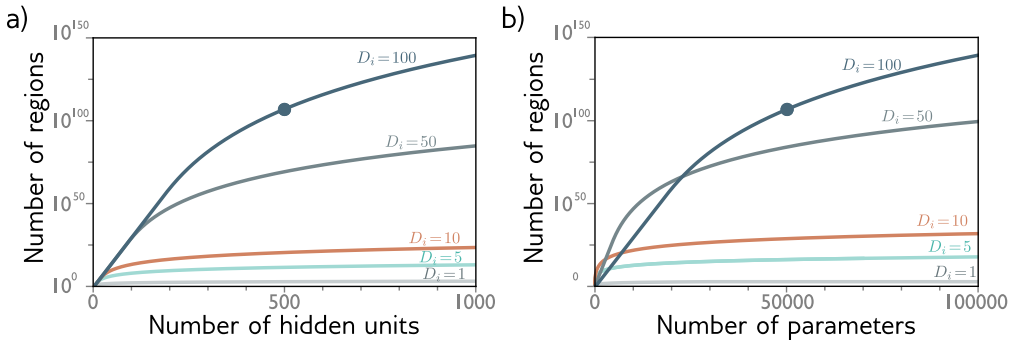

Figure 3.9 Linear regions vs. hidden units. a) Maximum possible regions as a function of the number of hidden units for five different input dimensions $D_i = \{1, 5, 10, 50, 100\}$. The number of regions increases very rapidly in high dimensions; with $D = 500$ units and input size $D_i = 100$, there can be greater than 10^{107} regions (solid circle). b) The same data plotted as a function of the number of parameters. The solid circle represents the same model as in panel (a) with $D = 500$ hidden units. This network has $51,001$ parameters and would be considered very small by modern standards.

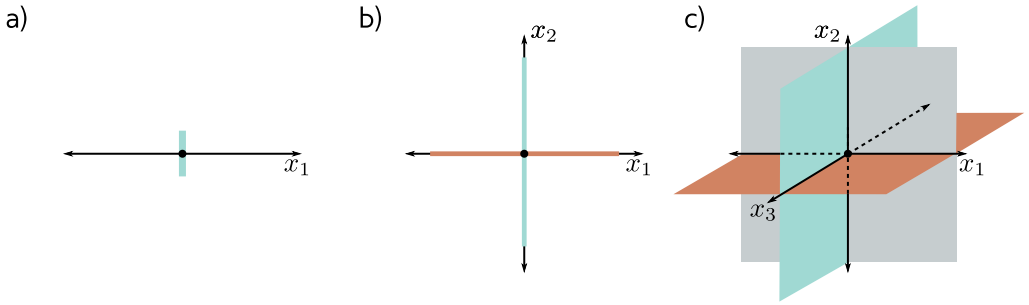

Figure 3.10 Number of linear regions vs. input dimensions. a) With a single input dimension, a model with one hidden unit creates one joint, which divides the axis into two linear regions. b) With two input dimensions, a model with two hidden units can divide the input space using two lines (here aligned with axes) to create four regions. c) With three input dimensions, a model with three hidden units can divide the input space using three planes (again aligned with axes) to create eight regions. Continuing this argument, it follows that a model with D_i input dimensions and D_i hidden units can divide the input space with D_i hyperplanes to create 2^{D_i} linear regions.

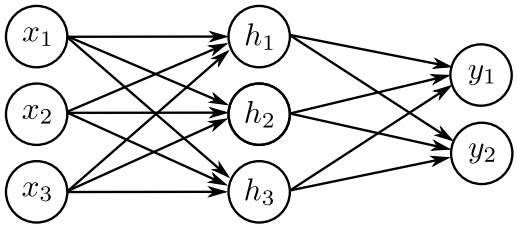

Figure 3.11 Visualization of neural network with three inputs and two outputs. This network has twenty parameters. There are fifteen slopes (indicated by arrows) and five offsets (not shown).

$$y_j = \phi_{j0} + \sum_{d=1}^{D} \phi_{jd} h_d, \tag{3.11}$$

where $a[\bullet]$ is a nonlinear activation function. The model has parameters $\boldsymbol{\phi} = \{\theta_{\bullet\bullet}, \phi_{\bullet\bullet}\}$. Figure 3.11 shows an example with three inputs, three hidden units, and two outputs.

Problems 3.14-3.17

The purpose of the activation function is to permit the model to describe nonlinear relations between the input and the output, and as such it must be nonlinear itself; with no activation function, or a linear activation function, the overall mapping from input to output would be restricted to be linear.

Many different activation functions have been tried (figure 3.13), but the most common choice is the ReLU. This has the merit of being easily interpretable. With ReLU activations, the network divides the input space into convex polytopes that are defined by the intersections of hyperplanes computed by the "joints" in the ReLU functions. Each of these convex polytopes contains a different linear function. The polytopes are the same for each of the D_o outputs, but the linear functions that they contain can differ.

3.5 Terminology

We conclude this chapter by introducing some terminology. Regrettably, neural networks have a lot of associated jargon. They are often referred to in terms of *layers*. The left of figure 3.12 is the *input layer*, the center is the *hidden layer*, and to the right is the *output layer*. We would say that the network in figure 3.12 has one hidden layer containing four hidden units. The hidden units themselves are sometimes referred to as *neurons*. When we pass data through the network, the values of the inputs to the hidden layer (i.e., before the ReLU functions are applied) are termed *pre-activations*. The values at the hidden layer (i.e., after the ReLU functions) are termed *activations*.

For historical reasons, any neural network with at least one hidden layer is also called a *multi-layer perceptron* or *MLP* for short. Networks with one hidden layer (as described in this chapter) are sometimes referred to as *shallow neural networks*. Networks with multiple hidden layers (as described in the next chapter) are referred to as *deep neural networks*. Neural networks in which the connections form an acyclic graph (i.e., a graph with no loops, as in all the examples in this chapter) are referred to as *feed-forward networks*. If every element in one layer connects to every element in the next (as in all the examples in this chapter), the network is *fully connected*. These connections

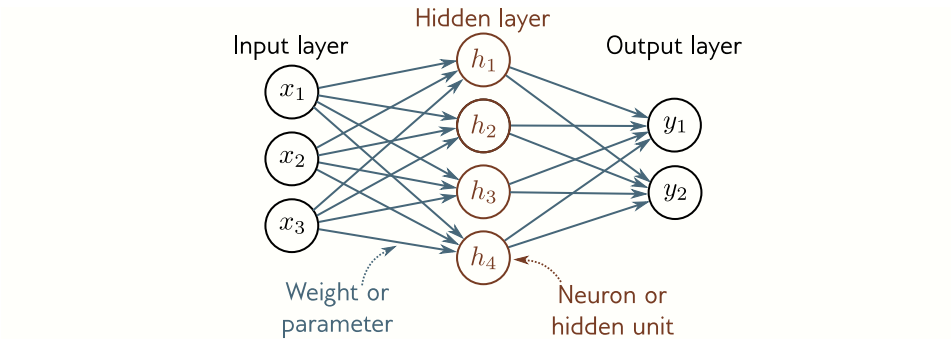

Figure 3.12 Terminology. A shallow network consists of an input layer, a hidden layer, and an output layer. Each layer is connected to the next by forward connections (arrows). For this reason, these models are referred to as feed-forward networks. When every variable in one layer connects to every variable in the next, we call this as a fully connected network. Each connection represents a slope parameter in the underlying equation and these parameters are termed weights. The variables in the hidden layer are termed neurons or hidden units. The values feeding into the hidden units are termed pre-activations and the values at the hidden units (i.e., after the ReLU function is applied) are termed activations.

represent slope parameters in the underlying equations and are referred to as *network weights*. The offset parameters (not shown in figure 3.12) are referred to as *biases*.

3.6 Summary

Shallow neural networks have one hidden layer. They (i) compute several linear functions of the input, (ii) pass each result through an activation function, and then (iii) take a linear combination of these activations to form the outputs. Shallow neural networks make predictions $\mathbf{y}$ based on inputs $\mathbf{x}$ by dividing the input space into a continuous surface of piecewise linear regions. With enough hidden units (neurons), shallow neural networks can approximate any continuous function to arbitrary precision.

The next chapter discusses deep neural networks, which extend the models from this chapter by adding more hidden layers. Chapters 5–7 describe how to train these models.

Notes

'Neural" networks: If the models in this chapter are just functions, why are they called "neural networks"? The connection is unfortunately tenuous. Visualizations like figure 3.12,

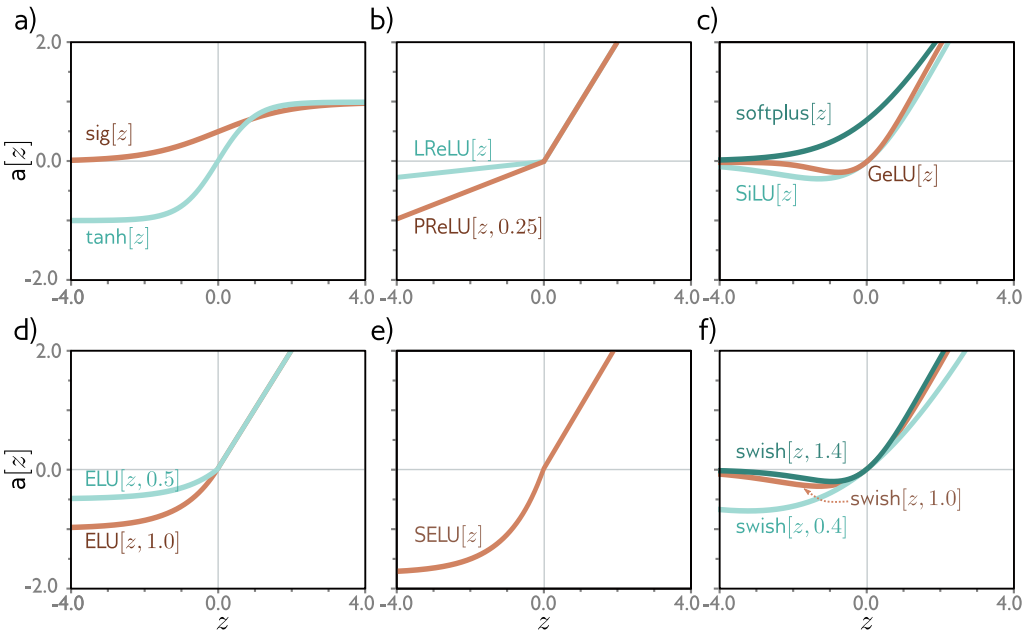

Figure 3.13 Activation functions. a) Logistic sigmoid and tanh functions. b) Leaky ReLU and parametric ReLU with parameter 0.25. c) SoftPlus, Guassian error linear unit, and sigmoid linear unit. d) Exponential linear unit with parameters 0.5 and 1.0, e) Scaled exponential linear unit. f) Swish with parameters 0.4, 1.0, and 1.4.

consist of a set of nodes (inputs, hidden units, and outputs) that are densely connected to one another. This bears a superficial similarity to neurons in the mammalian brain, which also have dense connections. However, there is scant evidence that brain computation works in the same way as neural networks, and it is unhelpful to think about biology going forward.

History of neural networks: McCulloch & Pitts (1943) first came up with the notion of an artificial neuron that combined inputs to produce an output, but this model did not have a practical learning algorithm. Rosenblatt (1958) developed the *perceptron*, which linearly combined inputs and then thresholded them to make a yes/no decision. He also provided an algorithm to learn the weights from data. Minsky & Papert (1969) argued that the linear function was inadequate for general classification problems, but that adding hidden layers with nonlinear activation functions (hence the term multi-layer perceptron) could allow the learning of more general input/output relations. However, they concluded that Rosenblatt's algorithm could not learn the parameters of such models. It was not until the 1980s that a practical algorithm (backpropagation, see chapter 7) was developed and significant work on neural networks resumed. The history of neural networks is chronicled by Kurenkov (2020) and Sejnowski (2018).

Activation functions: The ReLU function has been used as far back as Fukushima (1969). However, in the early days of neural networks, it was more common to use the logistic sigmoid or tanh activation functions (figure 3.13a). The ReLU was re-popularized by Jarrett et al. (2009), Nair & Hinton (2010), and Glorot et al. (2011) and is an important part of the success story of

modern neural networks. It has the nice property that the derivative of the output with respect to the input is always one for inputs greater than zero. This contributes to the stability and efficiency of training (see chapter 7) and contrasts with the derivatives of the earlier sigmoid activation functions, which saturate (become close to zero) for large inputs.

However, the ReLU function has the disadvantage that its derivative is zero for negative inputs. If all the training examples produce negative inputs to a given ReLU function, then we cannot improve the parameters feeding into this during training. The gradient with respect to the incoming weights is locally flat and so we cannot "walk downhill". This is known as the *dying ReLU* problem. To resolve this problem, many variations on the ReLU have been proposed (figure 3.13b) including the (i) the leaky ReLU (Maas et al. 2013), which also has a linear output for negative values with a smaller slope of 0.1, (ii) the parametric ReLU (He et al. 2015), which treats the slope of the negative portion as an unknown parameter and (iii) the concatenated ReLU (Shang et al. 2016), which produces two outputs, one of which clips below zero (i.e., like a typical ReLU) and one of which clips above zero.

A variety of continuous functions have also been investigated (figure 3.13c-d), including the softplus function (Glorot et al. 2011), Gaussian error linear unit (Hendrycks & Gimpel 2016), sigmoid linear unit (Hendrycks & Gimpel 2016), and exponential linear unit (Clevert et al. 2015). Most of these are attempts to avoid the dying neuron problem while limiting the gradient for negative values. Klambauer et al. (2017) introduced the scaled exponential linear unit (figure 3.13e), which is particularly interesting as it helps stabilize the variance of the activations when the input variance has a limited range (see section 7.3). Ramachandran et al. (2017) adopted an empirical approach to choosing an activation function. They searched the space of possible functions to find the one that performed best over a variety of supervised learning tasks. The optimal function was found to be $a[x] = x/(1+\exp[-\beta x])$ where β is a learned parameter (figure 3.13f). They termed this function *Swish*. Howard et al. (2019) approximated Swish by the HardSwish function, which has a very similar shape, but is faster to compute:

$$\text{HardSwish}[z] = \begin{cases} 0 & z < -3 \\ z(z+3)/6 & -3 \leq z \leq 3 \\ z & z > 3 \end{cases}. \tag{3.12}$$

There is no definitive answer as to which of these activations functions is empirically superior, although the leaky ReLU, parameterized ReLU, and many of the continuous functions can be shown to provide minor performance gains over the ReLU in particular situations. We restrict attention to neural networks with the basic ReLU function for the rest of this book because it's easy to characterize the functions that they create in terms of the number of linear regions.

Universal approximation theorem: The *width version* of this theorem states that a network with one hidden layer containing a finite number of hidden units and an activation function can approximate any continuous function on a compact subset of $\mathbb{R}^n$ to arbitrary accuracy. This was proved by Cybenko (1989) for sigmoid activations and was later shown to be true for a larger class of nonlinear activation functions (Hornik 1991).

Number of linear regions: Consider a shallow network with $D_i \geq 2$ dimensional inputs and D hidden units. The number of linear regions is determined by the intersections of the D hyperplanes created by the "joints" in the ReLU functions (e.g., figure 3.8d–f). Each region is created by a different combination of the ReLU functions clipping or not clipping the input. The number of regions created by D hyperplanes in the $D_i \leq D$ dimensional input space was shown by Zaslavsky (1975) to be at most $\sum_{j=0}^{D_i} \binom{D}{j}$. As a rule of thumb, shallow neural networks almost always have a larger number D of hidden units than input dimensions D_i and create a number of linear regions that is between 2^D and 2^{D_i}.

Problem 3.18

Linear, affine, and nonlinear functions: Technically, a linear transformation f[•] is any function that obeys the principle of superposition so $\text{f}[a+b] = \text{f}[a] + \text{f}[b]$. This definition implies that $\text{f}[2a] = 2\text{f}[a]$. The weighted sum $f[h_1, h_2, h_3] = \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$ is linear, but once the offset (bias) is added so $f[h_1, h_2, h_3] = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$, this is no longer true. To see this, consider that when we double the arguments of the former function, the output is doubled. This is not the case for the latter function, which is actually termed an *affine* function. However, it is common in machine learning to conflate these terms and we follow this convention here and refer to both as linear. All other functions we will encounter are nonlinear.

Problems

Problem 3.1 What kind of mapping from input to output would be created if the activation function in equation 3.1 was linear so that $a[z] = \psi_0 + \psi_1[z]$? What kind of mapping would be created if the activation function was removed entirely, so $a[z] = z$?

Problem 3.2 For each of the four linear regions in figure 3.3j, indicate which hidden units are inactive and which are active (i.e., which do and do not clip their inputs).

Problem 3.3 Derive expressions for the positions of the "joints" in function in figure 3.3j in terms of the ten parameters $\boldsymbol{\phi}$. Derive expressions for the slopes of the four linear regions.

Problem 3.4 Draw a version of figure 3.3 where the intercept and slope of the third hidden unit have changed as in figure 3.14c. Assume that the remaining parameters remain the same.

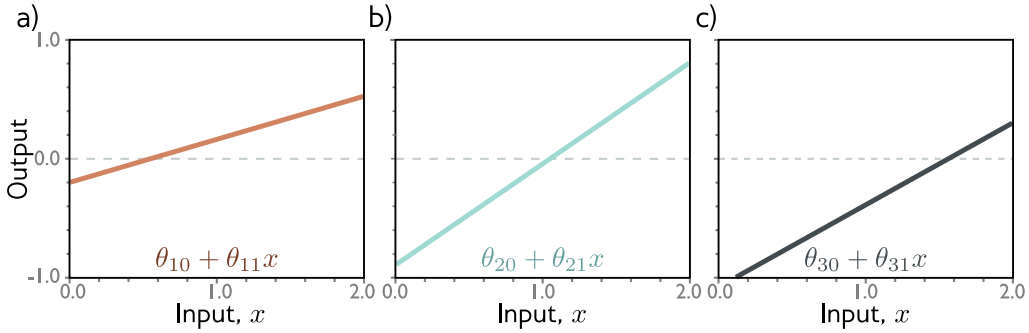

Figure 3.14 Processing in network with one input, three hidden units, and one output for problem 3.4. a–c) The input to each hidden unit is a linear function of the inputs. The first two are the same as in figure 3.3 but the last one differs.

Problem 3.5 Prove that the following property holds for non-negative $z \in \mathbb{R}^+$:

$$\text{ReLU}[\alpha z] = \alpha \text{ReLU}[z]. \tag{3.13}$$

This is known as the *non-negative homogeneity* property of the ReLU function.

Problem 3.6 Following on from problem 3.5, what happens to the shallow network defined in equations 3.3 and 3.4 when we multiply the slopes $\theta_{11}, \theta_{21}, \theta_{31}$ by a positive constant α and divide the slopes ϕ_1, ϕ_2, ϕ_3 by the same parameter α? What happens if α is negative?

Problem 3.7 Consider fitting the model in equation 3.1 using a least squares loss function. Does this loss function have a unique minimum? i.e., is there is a single "best" set of parameters?

Problem 3.8 Consider replacing the ReLU activation function with (i) the Heaviside step function heaviside$[z]$, (ii) the hyperbolic tangent function $\tanh[z]$, (iii) the rectangular function rect$[z]$, and (iv) the sinusoidal function $\sin[z]$, where:

$$\text{heaviside}[z] = \begin{cases} 0 & z < 0 \\ 1 & z \geq 0 \end{cases}, \tag{3.14}$$

and

$$\text{rect}[z] = \begin{cases} 0 & z < 0 \\ 1 & 0 \leq z \leq 1 \\ 0 & z > 1 \end{cases}. \tag{3.15}$$

Redraw a version of figure 3.3 for each of these functions. Provide an informal description of the family of functions can be created by neural networks with one input, three hidden units, and one output for each activation function.

Problem 3.9 Show that for figure 3.3, the third linear region has a slope that is the sum of the slopes of first and fourth linear regions.

Problem 3.10 Consider a neural network with one input, one output, and three hidden units. The construction in figure 3.3 shows how this creates four linear regions. Under what circumstances could this network produce a function with fewer than four linear regions?

Problem 3.11 How many parameters does the model in figure 3.6 have?

Problem 3.12 How many parameters does the model in figure 3.7 have?

Problem 3.13 What is the activation pattern for each of the seven regions in figure 3.7? In other words, which hidden units are active (pass the input) and which are inactive (clip the input) for each region?

Problem 3.14 Write out the equations that define the network in figure 3.11. There should be three equations to compute the three hidden units from the inputs and two equations to compute the outputs from the hidden units.

Problem 3.15 What is the maximum possible number of 3D linear regions that can be created by the network in figure 3.11?

Problem 3.16 Write out the equations for a network with two inputs, four hidden units, and three outputs. Draw this model in the style of figure 3.11.

Problem 3.17 Equations 3.10 and 3.11 define a general neural network with D_i inputs, one hidden layer containing D hidden units, and D_o outputs. Find an expression for the number of parameters in the model in terms of D_i, D, and D_o.

Problem 3.18 Show that the maximum number of regions created by a shallow network with $D_i = 2$ dimensional input, $D_o = 1$ dimensional output and $D = 3$ hidden units is seven as in figure 3.8j. Use the result of Zaslavsky (1975) that the maximum number of regions created by partitioning a D_i-dimensional space with D hyperplanes is $\sum_{j=0}^{D_i} \binom{D}{j}$. What is the maximum number of regions if we add two more hidden units to this model so $D = 5$?

Chapter 4

Deep neural networks

The last chapter described shallow neural networks, which have a single hidden layer. This chapter introduces deep neural networks, which have more than one hidden layer. With ReLU activation functions, both shallow and deep networks describe piecewise linear mappings from input to output.

As the number of hidden units increases, shallow neural networks improve their descriptive power. With enough hidden units, they can describe arbitrarily complex functions in high dimensions. However, it turns out that for some functions, the required number of hidden units is impractically large. Deep networks have the advantage that they produce many more linear regions than shallow networks for a given number of parameters, and so from a practical standpoint they can be used to describe a broader family of functions.

4.1 Composing neural networks

To gain insight into the behavior of deep neural networks, we first consider composing two shallow networks, so the output of the first becomes the input to the second. Consider two shallow networks with three hidden units each (figure 4.1a). The first network takes an input x and returns output y and is defined by:

$$\begin{aligned} h_1 &= \mathrm{a}[\theta_{10} + \theta_{11}x] \\ h_2 &= \mathrm{a}[\theta_{20} + \theta_{21}x] \\ h_3 &= \mathrm{a}[\theta_{30} + \theta_{31}x], \end{aligned} \tag{4.1}$$

and

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3. \tag{4.2}$$

The second network takes y as input and returns y' and is defined by:

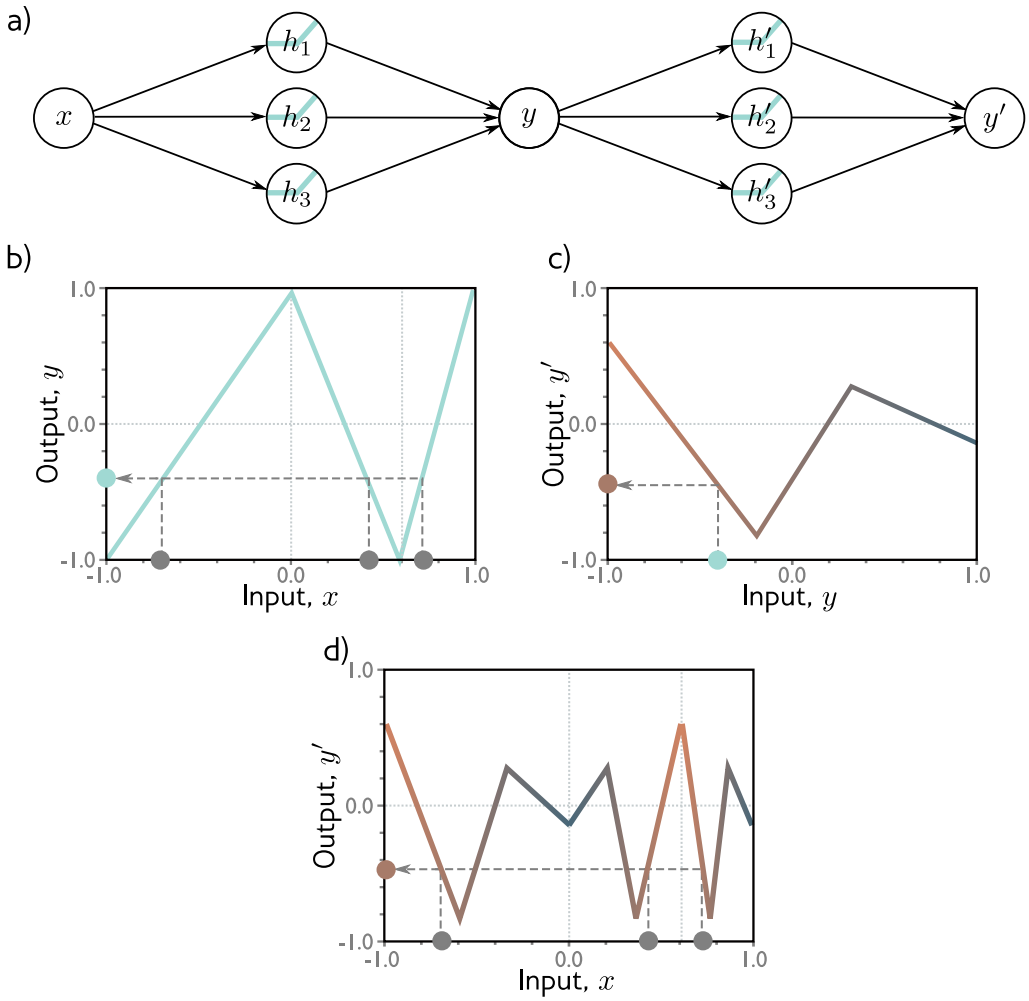

Figure 4.1 Composing two single-layer networks with three hidden units each. a) The output y of the first network constitutes the input to the second network. b) The first network maps inputs $x \in [0, 1]$ to outputs $y \in [0, 1]$ using a function comprised of three linear regions which are chosen so that they alternate the sign of their slope. Multiple inputs x (gray circles) now map to the same output y (cyan circle). c) The second network defines a function comprising three linear regions that takes y and returns y' (i.e., the cyan circle is mapped to the brown circle). d) The combined effect of these two functions when composed is that (i) three different inputs x are mapped to any given value of y by the first network and (ii) are processed in the same way by the second network; the result is that the function defined by the second network in panel (c) is duplicated three times, variously flipped and re-scaled according to the slope of the regions of panel (b).

$$\begin{aligned} h'_1 &= \text{a}[\theta'_{10} + \theta'_{11}y] \\ h'_2 &= \text{a}[\theta'_{20} + \theta'_{21}y] \\ h'_3 &= \text{a}[\theta'_{30} + \theta'_{31}y], \end{aligned} \tag{4.3}$$

and

$$y' = \phi'_0 + \phi'_1 h'_1 + \phi'_2 h'_2 + \phi'_3 h'_3. \tag{4.4}$$

With ReLU activations, this model also describes a family of piecewise linear functions. However, the number of linear regions is potentially greater than for a shallow network with six hidden units. To see this, consider choosing the first network to produce three alternating regions of positive and negative slope (figure 4.1b). This means that three different values of x are mapped to the same output y and the mapping from y to y' is applied three times. The overall effect is that the function defined by the second network is duplicated three times to create nine linear regions. The same principle applies in higher dimensions (figure 4.2).

Problem 4.1

A different way to think about composing networks is that the first network 'folds' the input space x back onto itself so that multiple inputs now generate the same output. Then the second network applies a function, which is replicated at all points that were folded on top of one another (figure 4.3).

4.2 From composing networks to deep networks

The previous section showed that we could create complex functions by passing the output of one shallow neural network into a second network. We'll now show that this is a special case of a deep network with two hidden layers.

The output of the first network ($y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$) is linear and so are the first operations of the second network (equation 4.3 in which we calculate $\theta'_{10} + \theta'_{11}y$, $\theta'_{20} + \theta'_{21}y$, and $\theta'_{30} + \theta'_{31}y$). Applying one linear function to another yields another linear function. Hence, when we substitute the expression for y into equation 4.3 the result is:

$$\begin{aligned} h'_1 &= \text{a}[\theta'_{10} + \theta'_{11}y] &&= \text{a}[\theta'_{10} + \theta'_{11}\phi_0 + \theta'_{11}\phi_1 h_1 + \theta'_{11}\phi_2 h_2 + \theta'_{11}\phi_3 h_3] \\ h'_2 &= \text{a}[\theta'_{20} + \theta'_{21}y] &&= \text{a}[\theta'_{20} + \theta'_{21}\phi_0 + \theta'_{21}\phi_1 h_1 + \theta'_{21}\phi_2 h_2 + \theta'_{21}\phi_3 h_3] \\ h'_3 &= \text{a}[\theta'_{30} + \theta'_{31}y] &&= \text{a}[\theta'_{30} + \theta'_{31}\phi_0 + \theta'_{31}\phi_1 h_1 + \theta'_{31}\phi_2 h_2 + \theta'_{31}\phi_3 h_3], \end{aligned} \tag{4.5}$$

which we can rewrite as:

$$\begin{aligned} h'_1 &= \text{a}[\psi_{10} + \psi_{11}h_1 + \psi_{12}h_2 + \psi_{13}h_3] \\ h'_2 &= \text{a}[\psi_{20} + \psi_{21}h_2 + \psi_{22}h_2 + \psi_{23}h_3] \\ h'_3 &= \text{a}[\psi_{30} + \psi_{31}h_2 + \psi_{32}h_2 + \psi_{33}h_3], \end{aligned} \tag{4.6}$$

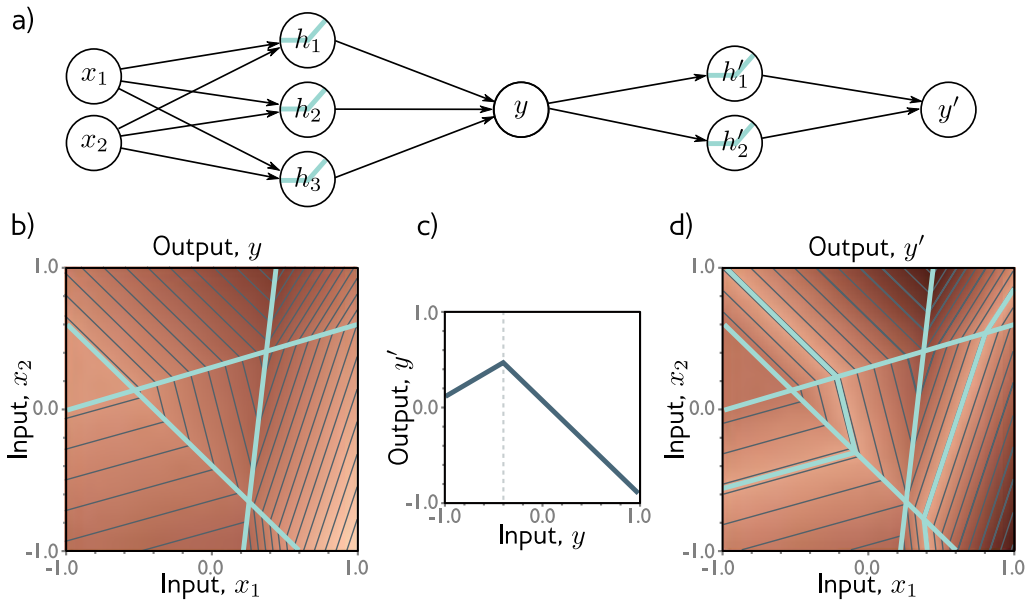

Figure 4.2 Composing neural networks with a 2D input. a) The first network (from figure 3.8) has three hidden units and takes two inputs x_1 and x_2 and returns a scalar output y. This is passed into a second network with two hidden units to produce y'. b) The first network produces a function consisting of seven linear regions, one of which is flat. c) The second network defines a function comprising two linear regions in $y \in [-1, 1]$. d) When composed together, each of the six non-flat regions from the first network is divided into two new regions by the second network to create a total of 13 linear regions.

where $\psi_{10} = \theta'_{10} + \theta'_{11}\phi_0, \psi_{11} = \theta'_{11}\phi_1, \psi_{12} = \theta'_{11}\phi_2$ and so on. The result is a network with two hidden layers (figure 4.4).

It follows that a network with two layers can represent the family of functions created by passing the output of one single-layer network into another. In fact, it represents a broader family, because in equation 4.6, the nine slope parameters $\psi_{11}, \psi_{21}, \ldots \psi_{33}$ can take arbitrary values, whereas in equation 4.5, these parameters are constrained to be the outer product $[\theta'_{11}, \theta'_{21}, \theta'_{31}]^T[\phi_1, \phi_2, \phi_3]$.

4.3 Deep neural networks

In the previous section we showed that when we compose two shallow networks, we get a special case of a deep network with two hidden layers. Now let's consider the general case of a deep network with two hidden layers, each containing three hidden units (figure 4.4).

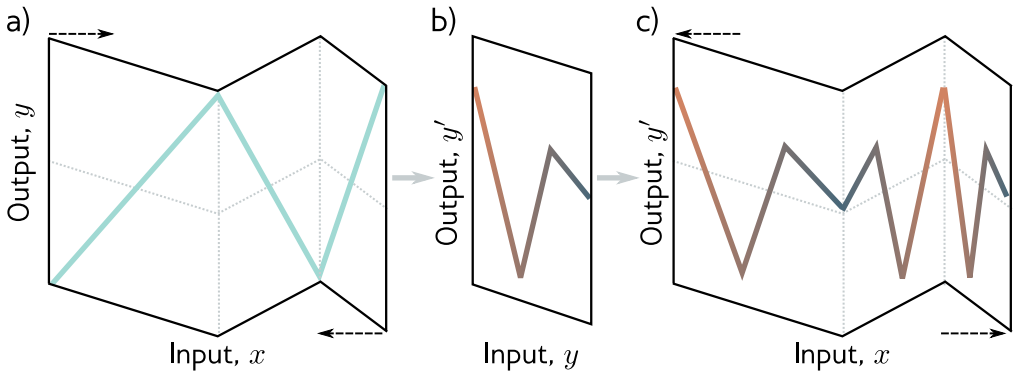

Figure 4.3 Deep networks as folding input space. a) One way to think about the first network from figure 4.1 is that it 'folds' the input space back on top of itself. b) The second network applies its function to the folded space. c) The final output is revealed by 'unfolding' again.

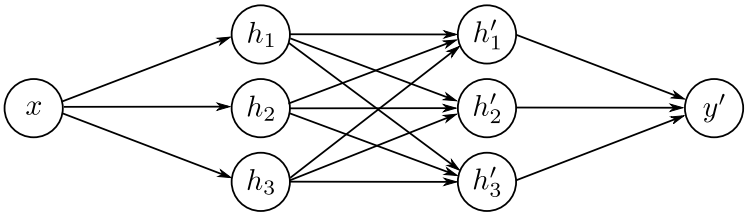

Figure 4.4 Neural network with one input, one output, and two hidden layers, each containing three hidden units.

The first layer is defined by:

$$\begin{aligned} h_1 &= \mathrm{a}[\theta_{10} + \theta_{11}x] \\ h_2 &= \mathrm{a}[\theta_{20} + \theta_{21}x] \\ h_3 &= \mathrm{a}[\theta_{30} + \theta_{31}x], \end{aligned} \tag{4.7}$$

the second layer by:

$$\begin{aligned} h'_1 &= \mathrm{a}[\psi_{10} + \psi_{11}h_1 + \psi_{12}h_2 + \psi_{13}h_3] \\ h'_2 &= \mathrm{a}[\psi_{20} + \psi_{21}h_1 + \psi_{22}h_2 + \psi_{23}h_3] \\ h'_3 &= \mathrm{a}[\psi_{30} + \psi_{31}h_1 + \psi_{32}h_2 + \psi_{33}h_3], \end{aligned} \tag{4.8}$$

and the output by:

$$y' = \phi'_0 + \phi'_1 h'_1 + \phi'_2 h'_2 + \phi'_3 h'_3. \tag{4.9}$$

Considering these equations leads to a different way to think about how the network constructs an increasingly complicated function (figure 4.5):

1. The three hidden units h_1, h_2, and h_3 in the first layer are computed as usual by forming linear functions of the input and passing these through ReLU activation functions (equation 4.7).
2. The pre-activations at the second layer are computed by taking three new linear functions of these hidden units (arguments of the activation functions in equation 4.8). At this point, we effectively have a shallow network with three outputs; we have computed three piecewise linear functions with the 'joints' between linear regions in the same places (see figure 3.6).
3. At the second hidden layer, another ReLU function $\mathrm{a}[\bullet]$ is applied to each function (equation 4.8), which clips them and adds new 'joints' to each.
4. The final output is a linear combination of these hidden units (equation 4.9).

In conclusion, we can either think of each layer as 'folding' the input space, or as creating new functions, which are clipped (creating new regions) and then recombined. Both descriptions provide partial insight into how deep neural networks operate. However, it's important not to lose sight of the fact that this is still merely an equation relating input x to output y'. Indeed, we can combine equations 4.7-4.9 to get a single expression:

$$\begin{aligned} y' = \; & \phi'_0 + \phi'_1 \mathrm{a}\left[\psi_{10} + \psi_{11}\mathrm{a}[\theta_{10} + \theta_{11}x] + \psi_{12}\mathrm{a}[\theta_{20} + \theta_{21}x] + \psi_{13}\mathrm{a}[\theta_{30} + \theta_{31}x]\right] \\ & + \phi'_2 \mathrm{a}[\psi_{20} + \psi_{21}\mathrm{a}[\theta_{10} + \theta_{11}x] + \psi_{22}\mathrm{a}[\theta_{20} + \theta_{21}x] + \psi_{23}\mathrm{a}[\theta_{30} + \theta_{31}x]] \\ & + \phi'_3 \mathrm{a}[\psi_{30} + \psi_{31}\mathrm{a}[\theta_{10} + \theta_{11}x] + \psi_{32}\mathrm{a}[\theta_{20} + \theta_{21}x] + \psi_{33}\mathrm{a}[\theta_{30} + \theta_{31}x]], \end{aligned} \tag{4.10}$$

although this is admittedly rather difficult to understand.

4.3.1 Hyperparameters

We can extend the deep network construction to more than two hidden layers; modern networks might have more than a hundred layers with thousands of hidden units at each layer. The number of hidden units in each layer is referred to as the *width* of the network, and the number of hidden layers as the *depth*. The total number of hidden units is a measure of the *capacity* of the network.

We denote the number of layers as K and the number of hidden units in each layer as $D_1, D_2, \ldots, D_K$. These are examples of *hyperparameters*. They are quantities that are chosen before we learn the parameters of the model (i.e., the slope and intercept terms). For fixed hyperparameters (e.g., $K = 2$ layers with $D_k = 3$ hidden units in each),

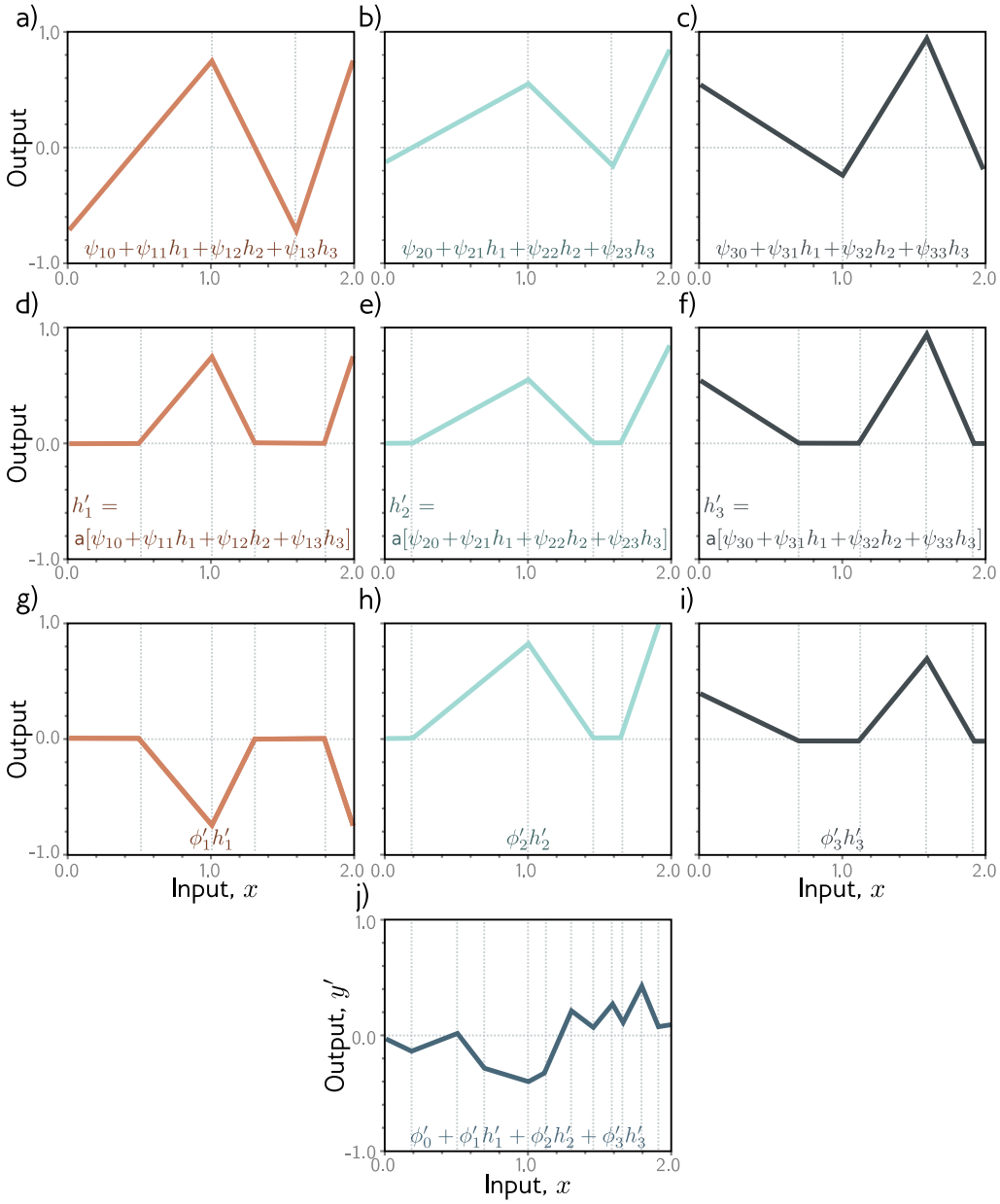

Figure 4.5 Computation for the deep network in figure 4.4. a-c) The inputs to the second hidden layer (i.e., the pre-activations) are three piecewise linear functions where the "joints" between the linear regions are at the same places (see figure 3.6). d-f) Each piecewise linear function is clipped to zero by the ReLU activation function. g-i) These clipped functions are then weighted with parameters ϕ'_1, ϕ'_2, and ϕ'_3 respectively. h) Finally, the clipped and weighted functions are summed together and an offset ϕ'_0 that controls the overall height is added.

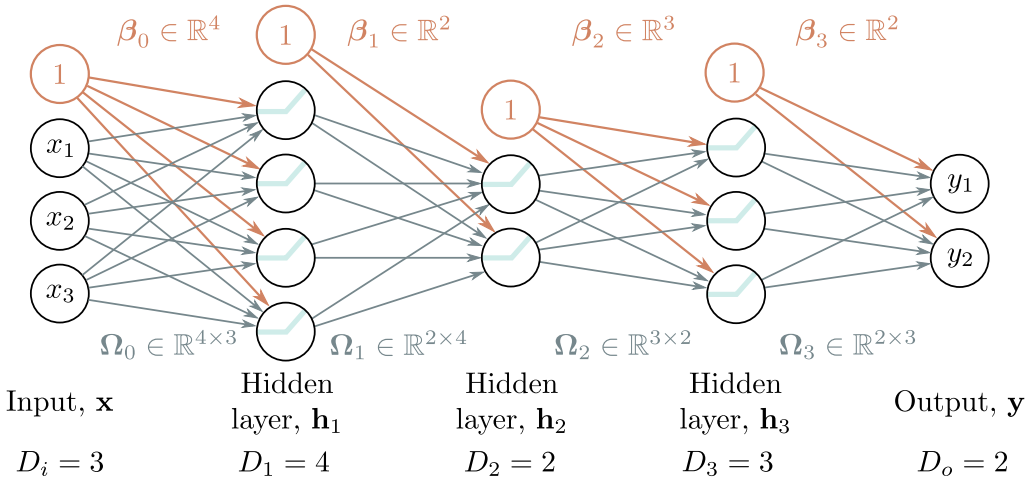

Figure 4.6 Matrix notation for network with $D_i = 3$ dimensional input $\mathbf{x}$, $D_o = 2$ dimensional output $\mathbf{y}$, and $K = 3$ hidden layers $\mathbf{h}_1, \mathbf{h}_2$ and $\mathbf{h}_3$ of dimensions $D_1 = 4$, $D_2 = 2$, and $D_3 = 3$ respectively. The weights are stored in matrices $\boldsymbol{\Omega}_k$ that pre-multiply the activations from the preceding layer to create the pre-activations at the subsequent layer. For example, the weight matrix $\boldsymbol{\Omega}_1$ that computes the pre-activations at $\mathbf{h}_2$ from the activations at $\mathbf{h}_1$ is of dimension 2×4. It is applied to the four hidden units in layer one and creates the two inputs to the hidden units at layer two. The biases are stored in vectors $\boldsymbol{\beta}_k$ and have the dimension of the layer that they feed into. For example, the bias vector $\boldsymbol{\beta}_2$ is length three.

the model describes a family of functions, and the parameters determine the particular function. Hence, when we also consider the hyperparameters, we can think of neural networks as representing a family of families of functions relating input to output.

4.4 Matrix notation

We have seen that a deep neural network simply consists of linear transformations alternating with activation functions. We could equivalently describe equations 4.7-4.9 in matrix notation as:

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} = \mathbf{a}\left[\begin{bmatrix} \theta_{10} \\ \theta_{20} \\ \theta_{30} \end{bmatrix} + \begin{bmatrix} \theta_{11} \\ \theta_{21} \\ \theta_{31} \end{bmatrix} x\right], \tag{4.11}$$

$$\begin{bmatrix} h'_1 \\ h'_2 \\ h'_3 \end{bmatrix} = \mathbf{a}\left[\begin{bmatrix} \psi_{10} \\ \psi_{20} \\ \psi_{30} \end{bmatrix} + \begin{bmatrix} \psi_{11} & \psi_{12} & \psi_{13} \\ \psi_{21} & \psi_{22} & \psi_{23} \\ \psi_{32} & \psi_{32} & \psi_{33} \end{bmatrix}\begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix}\right], \tag{4.12}$$

and

$$y' = \phi'_0 + \begin{bmatrix} \phi'_1 & \phi'_2 & \phi'_3 \end{bmatrix}\begin{bmatrix} h'_1 \\ h'_2 \\ h'_3 \end{bmatrix}, \tag{4.13}$$

or even more compactly in matrix notation as:

$$\begin{aligned} \mathbf{h} &= \mathbf{a}\left[\boldsymbol{\theta}_0 + \boldsymbol{\Theta} x\right] \\ \mathbf{h}' &= \mathbf{a}\left[\boldsymbol{\psi}_0 + \boldsymbol{\Psi}\mathbf{h}\right] \\ y &= \boldsymbol{\phi}'_0 + \boldsymbol{\Phi}'\mathbf{h}', \end{aligned} \tag{4.14}$$

where in each case the function $\mathbf{a}[\bullet]$ applies the activation function separately to every element of its input.

4.4.1 General formulation

This notation becomes cumbersome for networks with many layers, and so from now on we will describe the vector of hidden units at layer k as $\mathbf{h}_k$, the vector of biases (intercepts) that contribute to hidden layer $k+1$ as $\boldsymbol{\beta}_k$, and the weights (slopes) that are applied to the k^{th} layer and contribute to the $(k+1)^{th}$ layer as $\boldsymbol{\Omega}_k$. A general deep network $\mathbf{y} = \mathrm{f}[\mathbf{x}, \boldsymbol{\phi}]$ with K layers can now be written as:

$$\begin{aligned} \mathbf{h}_1 &= \mathbf{a}[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0\mathbf{x}] \\ \mathbf{h}_2 &= \mathbf{a}[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1\mathbf{h}_1] \\ \mathbf{h}_3 &= \mathbf{a}[\boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2\mathbf{h}_2] \\ &\vdots \\ \mathbf{h}_K &= \mathbf{a}[\boldsymbol{\beta}_{K-1} + \boldsymbol{\Omega}_{K-1}\mathbf{h}_{K-1}] \\ \mathbf{y} &= \boldsymbol{\beta}_K + \boldsymbol{\Omega}_K\mathbf{h}_K, \end{aligned} \tag{4.15}$$

The parameters $\boldsymbol{\phi}$ of this model comprise all of these weight matrices and bias vectors $\boldsymbol{\phi} = \{\boldsymbol{\beta}_k, \boldsymbol{\Omega}_k\}_{k=0}^{K}$.

If the k^{th} layer has D_k neurons, then the bias vector $\boldsymbol{\beta}_{k-1}$ will be of size D_k. The last bias vector $\boldsymbol{\beta}_K$ has the size D_o of the output. The first weight matrix $\boldsymbol{\Omega}_0$ has size $D_1 \times D_i$ where D_i is the size of the input. The last weight matrix $\boldsymbol{\Omega}_K$ is $D_o \times D_K$, and the remaining matrices $\boldsymbol{\Omega}_k$ are $D_k \times D_{k-1}$ (figure 4.6).

We can equivalently write the network as a single function:

Problems 4.2-4.3

$$\mathbf{y} = \boldsymbol{\beta}_K + \boldsymbol{\Omega}_K \mathbf{a}\left[\boldsymbol{\beta}_{K-1} + \boldsymbol{\Omega}_{K-1}\mathbf{a}\left[\ldots \boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2 \mathbf{a}\left[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1 \mathbf{a}\left[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}\right]\right]\ldots\right]\right]. \tag{4.16}$$

4.5 Shallow vs. deep neural networks

In the previous chapter we discussed shallow networks (which have a single hidden layer), and here we have considered deep networks (which have multiple hidden layers). We now compare these models.

4.5.1 Ability to approximate different functions

In section 3.2 we argued that shallow neural networks can model any continuous function arbitrarily closely with enough capacity (hidden units). In this chapter, we saw that a deep network with two hidden layers can represent the composition of two shallow networks. If the first of these shallow networks has enough capacity and the second just computes the identity function, then this deep network can also approximate any continuous function arbitrarily closely. Both shallow and deep networks can approximate any function using piecewise linear regions.

Problem 4.4

4.5.2 Number of linear regions per parameter

A shallow network with one input, one output, and $D > 2$ hidden units can create up to $D + 1$ linear regions with non-zero slopes and is defined by $3D + 1$ parameters. A deep network with K layers of $D > 2$ hidden units can create a function with up to $(D+1)^K$ linear regions using $3D + 1 + (K - 1)D(D + 1)$ parameters.

Problems 4.5-4.8

Figure 4.7a shows how the maximum number of linear regions increases as a function of the number of parameters for networks mapping scalar input x to scalar output y. Deep neural networks create much more complex functions for a fixed parameter budget. This effect is magnified as the number of input dimensions D_i increases (figure 4.7b), although computing the maximum number of regions is less straightforward.

This seems attractive, but the flexibility of the functions is still limited by the number of parameters. Both shallow and deep networks can create extremely large numbers of linear regions, but these contain complex dependencies and symmetries. We glimpsed some of these when we considered deep networks as 'folding' the input space (figure 4.3). So, it's not clear that the much larger number of regions is an advantage unless (i) there are similar symmetries in the real-world functions that we wish to approximate, or (ii) we have reason to believe that the mapping from input to output really does involve a composition of simpler functions.

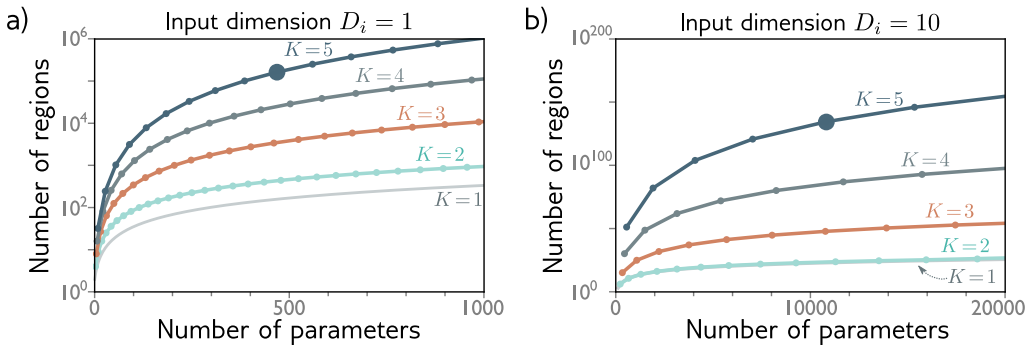

Figure 4.7 The maximum number of linear regions for neural networks increases very rapidly with the network depth. a) Network with $D_i = 1$ input. Each curve represents a fixed number of hidden layers K, as we vary the number of hidden units D per layer. For a fixed parameter budget (horizontal position), deeper networks produce more linear regions than shallower ones. A network with $K = 5$ layers and $D = 10$ hidden units per layer has 471 parameters (highlighted point) and can produce 161,051 regions. b) Network with $D_i = 10$ inputs. Here, a model with $K = 5$ layers and $D = 50$ hidden units per layer has 10,801 parameters (highlighted point) and can create more than 10^{134} linear regions.

4.5.3 Depth efficiency

Although both deep and shallow networks can model arbitrary functions, some functions can be approximated much more efficiently with deep networks. Functions have been identified that require a shallow network with exponentially more hidden units to achieve an equivalent approximation to a deep network. This phenomenon is referred to as the *depth efficiency* of neural networks. This property also seems attractive, but of course, it's still not clear that the real-world functions that we want to approximate fall into this category.

4.5.4 Large, structured inputs

We have discussed fully connected networks where every element of each layer contributes to every element of the subsequent one. However, these are not practical for large, structured inputs like images where the input might comprise $\sim 10^6$ pixels. The number of parameters would be prohibitive, and moreover, we want different parts of the image to be processed similarly; there is no point in independently learning to recognize the same object at every possible position in the image.

The solution is to process local image regions in parallel and then gradually integrate information from increasingly large regions. This kind of local-to-global processing is difficult to specify without using multiple layers (see chapter 10).

4.5.5 Training and generalization

A further reason why deep networks might be advantageous is the ease of fitting; it is easier to train moderately deep networks than shallow ones. It may be just that over-parameterized nets have a large family of roughly equivalent solutions that are easier to find. However, as depth increases further training becomes more difficult again (see chapter 11). Deep networks also seem to generalize to new data better than shallow ones. These phenomena are not well understood, and we return to them in chapter 19. In practice, the best results for most tasks have been achieved using networks with tens or hundreds of layers. A rare exception is networks for processing graphs (see chapter 13) where shallower architectures seem to perform better.

4.6 Summary

In this chapter, we first considered what happens when we compose two shallow networks. We argued that the first network 'folds' the input space and the second network then applies a piecewise linear function. The effects of the second network are duplicated where the input space is folded onto itself.

We then showed that this composition of shallow networks is a special case of a deep network with two layers. We interpreted the ReLU functions in each layer as clipping the input functions in multiple places, hence creating more and more 'joints' in the output function. We introduced the idea of hyperparameters, which for neural networks comprise the number of hidden layers and the number of hidden units in each.

Finally, we compared shallow and deep networks. We saw that (i) both networks can approximate any function given enough capacity, (ii) deep networks produce many more linear regions per parameter, (iii) some functions can be approximated much more efficiently by deep networks, (iv) large, structured inputs like images are best processed in a series of stages, and (v) in practice, the best results for most tasks are achieved using deep networks with many layers.

Now that we understand deep and shallow network models, we turn our attention to training them. In the next chapter, we discuss loss functions. For any given parameter values $\boldsymbol{\phi}$, the loss function returns a single number that indicates the mismatch between the model outputs and the ground truth predictions for a training dataset. In chapters 6 and 7, we deal with the training process itself, in which we seek the parameter values that minimize this loss.

Notes

Deep learning: It has long been understood that it is possible to build more complex functions by composing shallow neural networks or developing networks with more than one hidden layer. Indeed, the term 'deep learning' was first used by Dechter (1986). However, interest was limited

due to practical concerns; it was not possible to train such networks well. The modern era of deep learning was kick-started by startling improvements in image classification reported by Krizhevsky et al. (2012). This sudden progress was arguably due to the confluence of four factors: larger training datasets, improved processing power for training, the use of the ReLU activation function, and the use of stochastic gradient descent (see chapter 6). LeCun et al. (2015) present an overview of early advances in the modern era of deep learning.

Number of linear regions: For deep networks using a total of D hidden units with ReLU activations, the upper bound on the number of regions is 2^D (Montúfar et al. 2014). The same authors show that a deep ReLU network with D_i dimensional input, K layers each containing $D \geq D_i$ hidden units has $\mathcal{O}\left((D/D_i)^{(K-1)D_i} D^{D_i}\right)$ linear regions. Montúfar (2017), Arora et al. (2016) and Serra et al. (2018) all provide tighter upper bounds that consider the possibility that each layer has different numbers of neurons. Serra et al. (2018) provide an algorithm that counts the number of linear regions in a neural network, although it is only practical for very small networks.

If the number of hidden units D in each of the K layers is the same, and D is an integer multiple of the input dimensionality D_i, then the maximum number of linear regions N_r can be computed exactly and is:

$$N_r = \prod_{k=1}^{K-1} \left(\frac{D}{D_i} + 1\right)^{D_i} \sum_{j=0}^{D_i} \binom{D}{j}. \tag{4.17}$$

The first term in this expression corresponds to the first $K-1$ layers of the network, which can be thought of as repeatedly folding the input space. However, we now need to devote D/D_i neurons to each input dimension to create these folds. The last term in this equation (a sum of binomial coefficients) is the number of regions that can be created by a shallow network and is attributable to the last layer. For further information consult Montúfar et al. (2014), Pascanu et al. (2014), and Montúfar (2017).

Appendix C.8 binomial coefficient

Universal approximation theorem: We argued in section 4.5.1 that if the layers of a deep network have enough hidden units, then the width version of the universal approximation theorem applies: the network can approximate any continuous function on a compact subset of $\mathbb{R}^{D_i}$ to arbitrary accuracy. Lu et al. (2017) proved that a network with ReLU activation functions and at least $D_i + 4$ hidden units in each layer can approximate any D_i-dimensional Lebesgue integrable function to arbitrary accuracy given enough layers. This is known as the *depth version* of the universal approximation theorem.

Depth efficiency: There are several results that show that there are classes of functions that can be realized by deep networks but not by any shallow network whose capacity is bounded above exponentially. In other words, it would take an exponentially larger number of units in a shallow network to describe these functions accurately. This is known as the *depth efficiency* of neural networks.

Telgarsky (2016) shows that for any integer k, it is possible to construct networks with one input, one output, and $\mathcal{O}[k^3]$ layers of constant width, which cannot be realized with $\mathcal{O}[k]$ layers and less than 2^k width. Perhaps surprisingly, Eldan & Shamir (2015) showed that when there are multivariate inputs, there is a three-layer network that cannot be realized by any two-layer network if the capacity is sub-exponential in the input dimension. Cohen et al. (2015), Safran & Shamir (2016), and Poggio et al. (2016) also demonstrate functions that deep networks can approximate efficiently, but shallow ones cannot.

Width efficiency: Lu et al. (2017) pose the question of whether there are there wide shallow networks (i.e., shallow networks with lots of hidden units) that cannot be realized by narrow networks whose depth is not substantially larger. They show that there exist classes of wide,

shallow networks can only be expressed by narrow networks with polynomial depth. This is known as the *width efficiency* of neural networks. This polynomial lower bound on width is less restrictive than the exponential lower bound on depth, suggesting that depth is more important.

Problems

Problem 4.1 Consider composing the two neural networks in figure 4.8. Draw a plot of the relationship between the input x and output y' for $x \in [-1, 1]$.

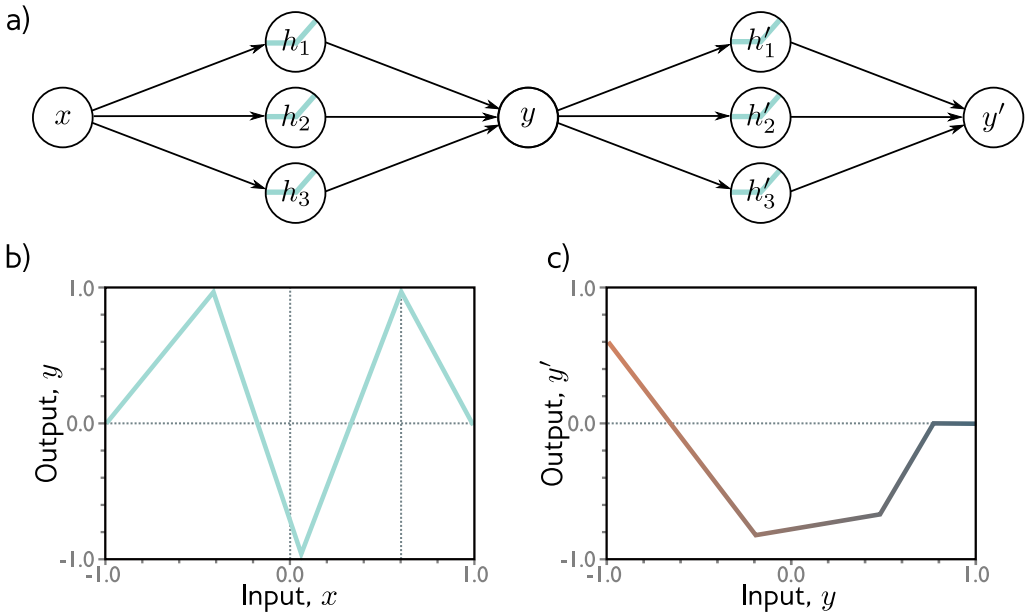

Figure 4.8 Composition of two networks for problem 4.1. a) Two networks are composed together so that the output y of the first network becomes the input to the second. b) The first network computes this function with output values $y \in [-1, 1]$. c) The second network computes this function on the input range $y \in [-1, 1]$.

Problem 4.2 Using the non-negative homogeneity property of the ReLU function (see problem 3.5), show that:

$$\text{ReLU}\left[\boldsymbol{\beta}_2 + \lambda_2 \cdot \boldsymbol{\Omega}_2 \left[\boldsymbol{\beta}_1 + \lambda_1 \boldsymbol{\Omega}_1 \mathbf{x}\right]\right] = \lambda_0 \cdot \lambda_1 \cdot \text{ReLU}\left[\frac{1}{\lambda_0 \cdot \lambda_1}\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1\left[\frac{1}{\lambda_0}\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}\right]\right], \quad (4.18)$$

where λ_1 and λ_2 are non-negative scalars. From this, we see that the weight matrices can be re-scaled by any magnitude as long as the biases are also adjusted, and the scale factors can be re-applied at the end of the network.

Problem 4.3 Write out the equations for a deep neural network that takes $D_i = 5$ inputs, $D_o = 4$ outputs and has three hidden layers of sizes $D_1 = 20$, $D_2 = 10$, and $D_3 = 7$ respectively in both the forms of equations 4.15 and 4.16. What are the sizes of each weight matrix $\boldsymbol{\Omega}_\bullet$ and bias vector $\boldsymbol{\beta}_\bullet$?

Problem 4.4 Choose values for the parameters $\boldsymbol{\phi} = \{\phi_0, \phi_1, \phi_2, \phi_3, \theta_{10}, \theta_{11}, \theta_{20}, \theta_{21}, \theta_{30}, \theta_{31}\}$ for the shallow neural network in equation 3.1 that will define an identity function over a finite range $x \in [a, b]$.

Problem 4.5 Figure 4.9 shows the activations in the three hidden units of a shallow network (as in figure 3.3). The slopes in the hidden units are 1.0, 1.0, and -1.0 respectively and the 'joints' in the hidden units are at positions 1/6, 2/6, and 4/6. Find values of $\phi_0, \phi_1, \phi_2, \phi_3$ that will combine the hidden unit activations as $\phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$ to create a function with four linear regions that oscillate between output values of zero and one. The slope of the left-most region should be positive, the next one negative, and so on. How many linear regions will we create if we compose this network with itself? How many will we create if we compose it with itself K times?

Problem 4.6 Following from problem 4.5, is it possible to create a function with three linear regions that oscillates back and forth between output values of zero and one using a shallow network with two hidden units? Is it possible to create a function with five linear regions that oscillates in the same way using a shallow network with four hidden units?

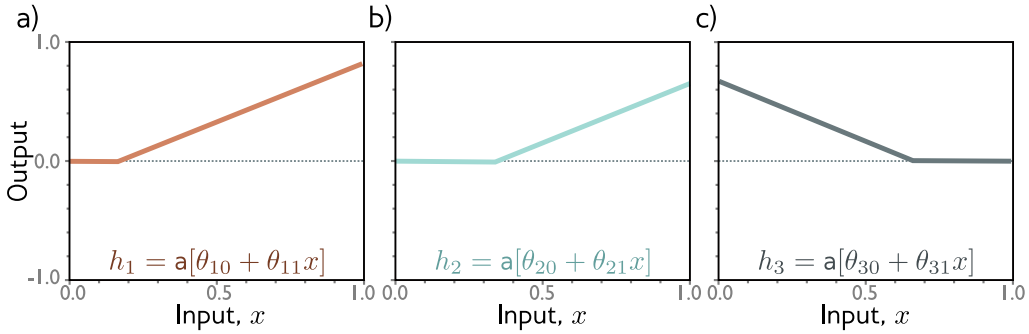

Figure 4.9 Hidden unit activations for problem 4.5. a) First hidden unit has a joint at position $x = 1/6$ and a slope of one in the active region. b) Second hidden unit has a joint at position $x = 2/6$ and a slope of one in the active region. c) Third hidden unit has a joint at position $x = 4/6$ and a slope of minus one in the active region.

Problem 4.7 Consider a deep neural network with a single input, a single output, and K hidden layers each of which contains D hidden units. Show that this network will have a total of $3D + 1 + (K - 1)D(D + 1)$ parameters.

Problem 4.8 Consider two neural networks that map a scalar input x to a scalar output y. The first network is shallow and has $D = 286$ hidden units. The second is deep and has $K = 10$

layers, each containing $D = 5$ hidden units. How many parameters does each network have? How many linear regions can each network make? Which do you think would run faster?

Chapter 5

Loss functions

The last three chapters described linear regression, shallow neural networks, and deep neural networks respectively. Each represents a family of functions that map input to output, where the particular member of the family is determined by the model parameters $\boldsymbol{\phi}$. When we train these models, we seek the parameters that produce the best possible mapping from input to output for the task we are considering. This chapter defines what is meant by the 'best possible' mapping.

That definition requires a training dataset $\{\mathbf{x}_i, \mathbf{y}_i\}$ of input/output pairs. A *loss function* or *cost function* $L[\boldsymbol{\phi}]$ returns a single number that describes the mismatch between the model predictions $\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ and their corresponding ground-truth outputs $\mathbf{y}_i$. During training, we seek parameter values $\boldsymbol{\phi}$ that minimize the loss, and hence map the training inputs to the outputs as closely as possible.

We saw one example of a loss function in chapter 2; the least squares loss function is suitable for univariate regression problems for which the target is a scalar $y \in \mathbb{R}$. It computes the sum of the squares of the deviations between the model predictions $\mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ and the true values y_i.

In this chapter, we provide a framework that both justifies the choice of the least squares criterion for scalar outputs and allows us to build loss functions for other prediction types. We consider *multivariate regression*, where the prediction is a vector $\mathbf{y} \in \mathbb{R}^D$, and *classification*, where the prediction is one of K categories $y \in [1, \ldots K]$. In the following two chapters, we address model training, in which the goal is to find the parameter values that minimize these loss functions.

5.1 Maximum likelihood

In this section, we develop a recipe for constructing loss functions. Consider a machine learning model $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ with parameters $\boldsymbol{\phi}$ that computes an output from input data $\mathbf{x}$. Until now, we have implied that the model directly computes a prediction $\mathbf{y}$. We now shift perspective and consider the model as computing a conditional probability distribution $Pr(\mathbf{y}|\mathbf{x})$ over possible outputs $\mathbf{y}$ given input $\mathbf{x}$. The loss function encourages

Appendix B.1.1
Conditional probability

each training output $\mathbf{y}_i$ to have high probability or *likelihood* under the distribution $Pr(\mathbf{y}_i|\mathbf{x}_i)$ computed from the corresponding input $\mathbf{x}_i$.

5.1.1 Computing a distribution over outputs

This raises the question of exactly how a model $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ can be adapted to compute a probability distribution. The solution is simple. First, we choose a parametric distribution $Pr(\mathbf{y}|\boldsymbol{\theta})$ that is defined on the output domain $\mathbf{y}$. Then we use the network to compute one or more of the distribution parameters $\boldsymbol{\theta}$.

For example, if the prediction domain is $y \in \mathbb{R}$, we might choose the univariate normal distribution, which is defined on $y \in \mathbb{R}$. This distribution is defined by the mean μ and variance σ^2, so $\boldsymbol{\theta} = \{\mu, \sigma^2\}$. In this case, the machine learning model $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ might predict the mean μ, and the variance σ^2 could be treated as an unknown constant.

5.1.2 Maximum likelihood criterion

The model now computes a different distribution parameter $\boldsymbol{\theta}_i$ for each training input $\mathbf{x}_i$. Each observed training output $\mathbf{y}_i$ should have high probability under its corresponding distribution $Pr(\mathbf{y}_i|\boldsymbol{\theta}_i)$. Hence, we choose the model parameters $\boldsymbol{\phi}$ so that they maximize the combined likelihood across all I training examples:

$$\begin{aligned}\hat{\boldsymbol{\phi}} &= \underset{\boldsymbol{\phi}}{\operatorname{argmax}} \left[\prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{x}_i)\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmax}} \left[\prod_{i=1}^{I} Pr(\mathbf{y}_i|\boldsymbol{\theta}_i)\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmax}} \left[\prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}])\right]. \end{aligned} \tag{5.1}$$

For obvious reasons, this is known as the *maximum likelihood* criterion.

Here we are implicitly making two assumptions. First, we assume that the data are identically distributed (the form of the probability distribution over the outputs $\mathbf{y}_i$ is the same for each data point). Second, we assume that the conditional distributions $Pr(\mathbf{y}_i|\mathbf{x}_i)$ of the output given the input are independent, so the total likelihood of the training data decomposes as:

Appendix B.1.4 Independence

$$Pr(\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_I|\mathbf{x}_1, \mathbf{x}_2, \ldots \mathbf{x}_I) = \prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{x}_i). \tag{5.2}$$

In other words, we are assuming the data are *independent and identically distributed* or *i.i.d.* for short.

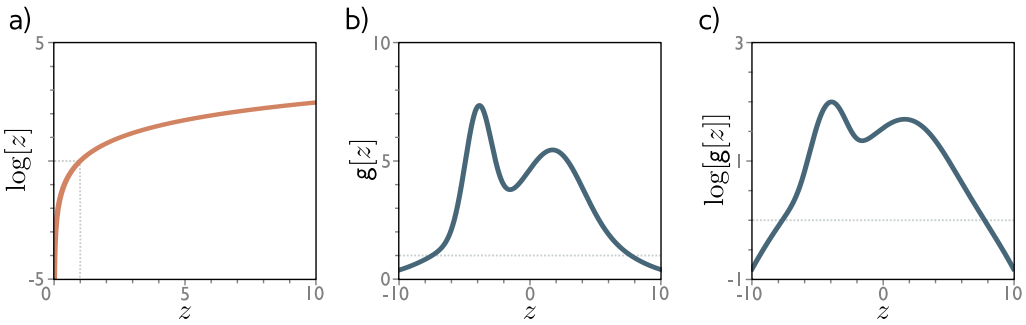

Figure 5.1 The log transform. a) The log function is monotonically increasing. If $z > z'$, then $\log[z] > \log[z']$. It follows that the maximum of any function $g[z]$ will be at the same position as the maximum of $\log[g[z]]$. b) A function $g[z]$. c) The logarithm of this function $\log[g[z]]$. All positions on $g[z]$ with a positive slope retain a positive slope after the log transform and those with a negative slope retain a negative slope. The position of the maximum remains the same.

5.1.3 Maximizing log-likelihood

The maximum likelihood criterion in equation 5.1 is not very practical. Each term $Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}])$ can be small, and so when we take the product of many of these terms, the result can be an extremely small number. It may be difficult to represent this quantity with finite precision math. Fortunately, we can equivalently maximize the logarithm of the likelihood:

$$\begin{aligned}\hat{\boldsymbol{\phi}} &= \underset{\boldsymbol{\phi}}{\operatorname{argmax}} \left[\prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}])\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmax}} \left[\log\left[\prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}])\right]\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmax}} \left[\sum_{i=1}^{I} \log\Big[Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}])\Big]\right]. \end{aligned} \tag{5.3}$$

This *log-likelihood* criterion is equivalent because the logarithm is a monotonically increasing function: if $z > z'$, then $\log[z] > \log[z']$ and vice versa (figure 5.1). This means that if we change the model parameters $\boldsymbol{\phi}$ in such a way that they improve the log-likelihood criterion, then we are also improving the original maximum likelihood criterion. It also follows that the overall maxima of the two criteria must be in the same place, and so the best model parameters $\hat{\boldsymbol{\phi}}$ are the same in both cases. However, the log-likelihood criterion has the practical advantage that it involves a sum of terms and not a product, and so representing it with finite precision isn't problematic.

5.1.4 Minimizing negative log-likelihood

Finally, we note that by convention, model fitting problems are framed in terms of minimization of a loss. To convert the maximum log-likelihood criterion to a minimization problem, we multiply by minus one, which gives us the *negative log-likelihood criterion*:

$$\begin{aligned}\hat{\boldsymbol{\phi}} &= \underset{\boldsymbol{\phi}}{\operatorname{argmax}}\left[\sum_{i=1}^{I}\log\Big[Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i,\boldsymbol{\phi}])\Big]\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[-\sum_{i=1}^{I}\log\Big[Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i,\boldsymbol{\phi}])\Big]\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\Big[\mathrm{L}[\boldsymbol{\phi}]\Big], \end{aligned} \tag{5.4}$$

which is what finally forms the loss function $\mathrm{L}[\boldsymbol{\phi}]$.

5.1.5 Inference

The network no longer directly predicts the outputs $\mathbf{y}$, but instead determines a probability distribution over $\mathbf{y}$. When we perform inference, we often want a point estimate rather than a distribution, and so we return the maximum of the distribution:

$$\hat{\mathbf{y}} = \underset{\mathbf{y}}{\operatorname{argmax}}\left[Pr(\mathbf{y}|\mathbf{f}[\mathbf{x},\boldsymbol{\phi}])\right]. \tag{5.5}$$

It is usually possible to find an expression for this in terms of the distribution parameters $\boldsymbol{\theta}$ predicted by the model. For example, in the univariate normal distribution, the maximum occurs at the mean μ.

5.2 Recipe for constructing loss functions

Let's summarize the recipe for constructing loss functions for training data $\{\mathbf{x}_i, \mathbf{y}_i\}$ using the maximum likelihood approach:

1. Choose a suitable probability distribution $Pr(\mathbf{y}|\boldsymbol{\theta})$ that is defined over the domain of the predictions $\mathbf{y}$ and has distribution parameters $\boldsymbol{\theta}$.
2. Set the machine learning model $\mathbf{f}[\mathbf{x},\boldsymbol{\phi}]$ to predict one or more of these parameters so $\boldsymbol{\theta} = \mathbf{f}[\mathbf{x},\boldsymbol{\phi}]$ and $Pr(\mathbf{y}|\boldsymbol{\theta}) = Pr(\mathbf{y}|\mathbf{f}[\mathbf{x},\boldsymbol{\phi}])$.
3. To train the model, find the network parameters $\hat{\boldsymbol{\phi}}$ that minimize the negative log-likelihood loss function over the training dataset pairs $\{\mathbf{x}_i, \mathbf{y}_i\}$:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[L[\boldsymbol{\phi}]\right] = \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[-\sum_{i=1}^{I}\log\Big[Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i,\boldsymbol{\phi}])\Big]\right]. \tag{5.6}$$

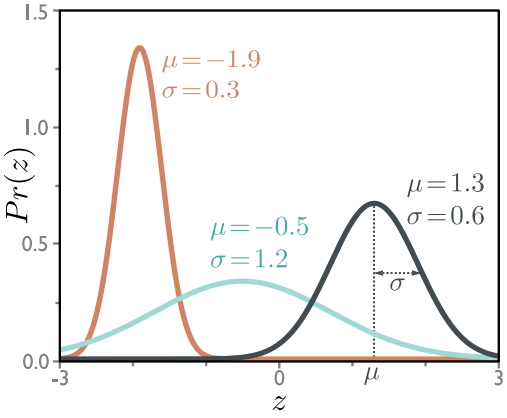

Figure 5.2 The univariate normal distribution (also known as the Gaussian distribution) is defined on the real line $z \in \mathbb{R}$ and has parameters μ and σ^2. The mean μ determines the position of the peak. The positive root of the variance σ^2 (the standard deviation) determines the width of the distribution. Since the total probability density sums to one, the peak becomes higher as the variance decreases and the distribution becomes narrower.

4. To perform inference for a new test example $\mathbf{x}$, return either the full distribution $Pr(\mathbf{y}|\mathbf{f}[\mathbf{x}, \hat{\boldsymbol{\phi}}])$ or the maximum of this distribution.

We devote most of the rest of this chapter to constructing loss functions for common prediction types using this recipe.

5.3 Example 1: univariate regression

We start by considering univariate regression models. Here the goal is to predict a single scalar output $y \in \mathbb{R}$ from input $\mathbf{x}$ using a model $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ with parameters $\boldsymbol{\phi}$. Following the recipe, we first choose a probability distribution over the output domain y. We select the univariate normal distribution (figure 5.2), which is defined over $y \in \mathbb{R}$. This distribution has two parameters (the mean μ and variance σ^2), and has probability density function:

$$Pr(y|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(y-\mu)^2}{2\sigma^2}\right]. \tag{5.7}$$

Second, we set the machine learning model $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ to compute one or more of the parameters of the output distribution. In this case, we will just compute the mean, so that $\mu = \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$:

$$Pr(y|\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}], \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(y-\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}])^2}{2\sigma^2}\right]. \tag{5.8}$$

Our goal is to find the parameters $\boldsymbol{\phi}$ that make the training data $\{\mathbf{x}_i, y_i\}$ most likely given this distribution (figure 5.3). To accomplish this, we choose a loss function $L[\boldsymbol{\phi}]$ based on the negative log-likelihood:

$$\begin{aligned} L[\boldsymbol{\phi}] &= -\sum_{i=1}^{I} \log\left[Pr(y_i|\mathrm{f}[\mathbf{x}_i,\boldsymbol{\phi}],\sigma^2)\right] \\ &= -\sum_{i=1}^{I} \log\left[\frac{1}{\sqrt{2\pi\sigma^2}}\exp\left[-\frac{(y_i-\mathrm{f}[\mathbf{x}_i,\boldsymbol{\phi}])^2}{2\sigma^2}\right]\right]. \end{aligned} \tag{5.9}$$

When we train the model, we seek parameters $\hat{\boldsymbol{\phi}}$ that minimize this loss.

5.3.1 Least squares loss function

Now let's perform some algebraic manipulations on the loss function. We seek:

$$\begin{aligned} \hat{\boldsymbol{\phi}} &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[-\sum_{i=1}^{I} \log\left[\frac{1}{\sqrt{2\pi\sigma^2}}\exp\left[-\frac{(y_i-\mathrm{f}[\mathbf{x}_i,\boldsymbol{\phi}])^2}{2\sigma^2}\right]\right]\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[-\sum_{i=1}^{I} \log\left[\frac{1}{\sqrt{2\pi\sigma^2}}\right] - \frac{(y_i-\mathrm{f}[\mathbf{x}_i,\boldsymbol{\phi}])^2}{2\sigma^2}\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[-\sum_{i=1}^{I} -\frac{(y_i-\mathrm{f}[\mathbf{x}_i,\boldsymbol{\phi}])^2}{2\sigma^2}\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[\sum_{i=1}^{I}(y_i-\mathrm{f}[\mathbf{x}_i,\boldsymbol{\phi}])^2\right], \end{aligned} \tag{5.10}$$

where we have removed the first term between the second and third line because it does not depend on $\boldsymbol{\phi}$. We have removed the denominator between the third and fourth lines, as this is just a constant scaling factor that does not affect the position of the minimum.

The result of these manipulations is the least squares loss function that we originally introduced when we discussed linear regression in chapter 2:

$$L[\boldsymbol{\phi}] = \sum_{i=1}^{I}(y_i-\mathrm{f}[\mathbf{x}_i,\boldsymbol{\phi}])^2. \tag{5.11}$$

We see that the least squares loss function follows naturally from the assumptions that the prediction errors are (i) independent and (ii) drawn from a normal distribution with mean $\mu = \mathrm{f}[\mathbf{x}_i,\boldsymbol{\phi}]$ (figure 5.3).

5.3.2 Inference

The network no longer directly predicts y but instead predicts the mean $\mu = \mathrm{f}[\mathbf{x},\boldsymbol{\phi}]$ of the normal distribution over y. When we perform inference, we usually want a single

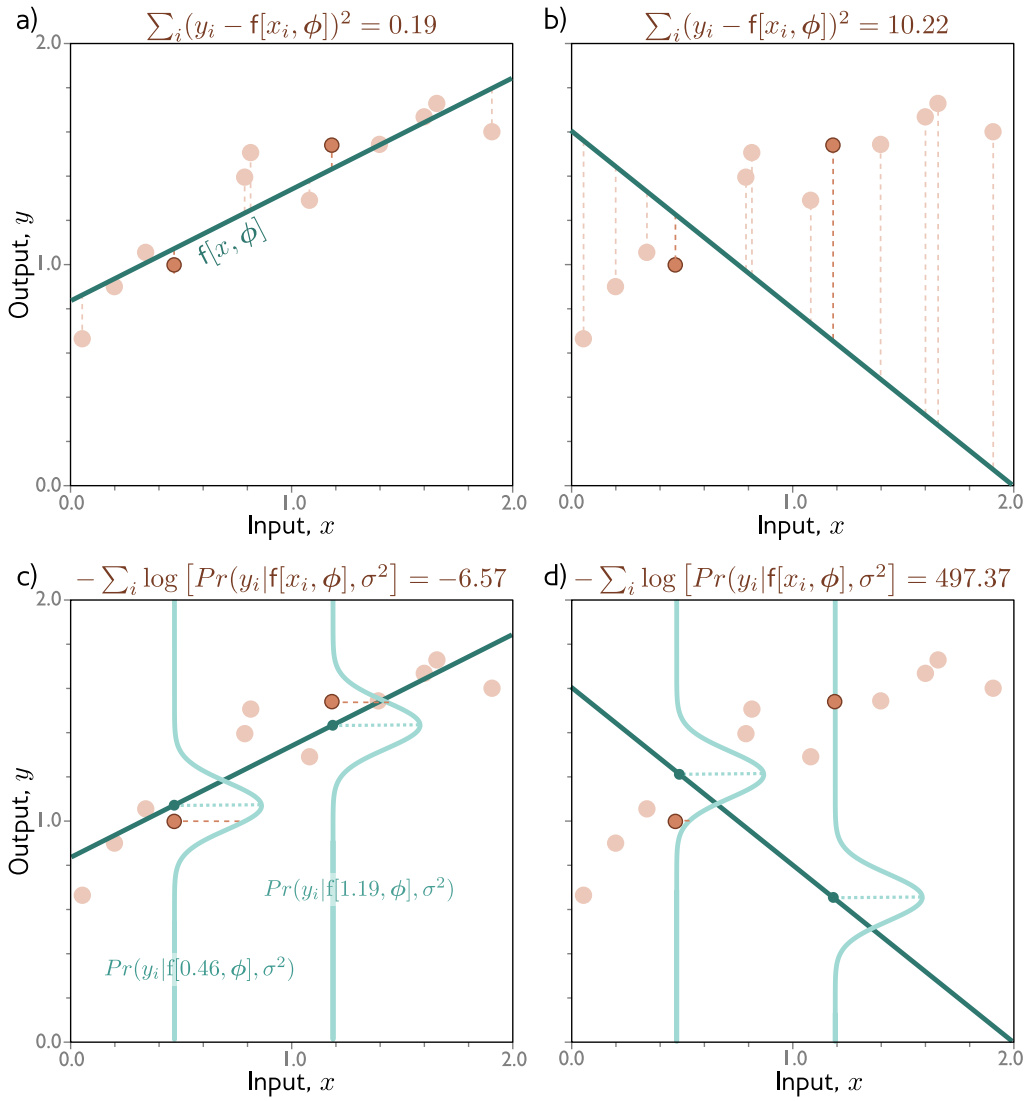

Figure 5.3 Equivalence of least squares and maximum likelihood loss for the normal distribution. a) Consider the linear model from figure 2.2. The least squares criterion minimizes the sum of the squares of the deviations (dashed lines) between the model prediction $\mathrm{f}[x_i, \boldsymbol{\phi}]$ (green line) and the true output values y_i (orange dots). Here the fit is good, so these deviations are small (e.g., for the two highlighted points). b) For these parameters, the fit is bad and the squared deviations are large. c) The least squares criterion follows from the assumption that the model predicts the mean of a normal distribution over the outputs and that we maximize the probability. For the first case, the model fits well, so the probability $Pr(y_i|x_i)$ of the data (horizontal orange dashed lines) is large (and the negative log probability is small). d) For the second case, the model fits badly, so the probability is small (and the negative log probability is large).

"best" point estimate $\hat{y}$ and take the maximum of the predicted distribution:

$$\hat{y} = \underset{y}{\operatorname{argmax}} \left[Pr(y|\mathbf{f}[\mathbf{x}, \hat{\boldsymbol{\phi}}]) \right]. \tag{5.12}$$

For the univariate normal, the maximum position is determined by the mean parameter μ (figure 5.2). This is exactly what the model computed, and so $\hat{y} = \mathrm{f}[\mathbf{x}, \hat{\boldsymbol{\phi}}]$.

5.3.3 Estimating variance

To formulate the least squares loss function, we assumed that the network predicted the mean of a normal distribution. The final expression in equation 5.11 (perhaps surprisingly) does not depend on the variance σ^2. However, there is nothing to stop us from treating σ^2 as a parameter of the model and minimizing equation 5.9 with respect to both the model parameters $\boldsymbol{\phi}$ and the distribution variance σ^2:

$$\hat{\boldsymbol{\phi}}, \hat{\sigma}^2 = \underset{\boldsymbol{\phi}, \sigma^2}{\operatorname{argmin}} \left[-\sum_{i=1}^{I} \log \left[\frac{1}{\sqrt{2\pi\sigma^2}} \exp \left[-\frac{(y_i - \mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}])^2}{2\sigma^2} \right] \right] \right]. \tag{5.13}$$

In inference, the model predicts the mean $\mu = \mathrm{f}[\mathbf{x}, \hat{\boldsymbol{\phi}}]$ from the input, and we learned the variance σ^2 during the training process. The former is the best prediction. The latter tells us about the uncertainty of the prediction.

5.3.4 Heteroscedastic regression

The model above assumes that the variance of the data is constant everywhere. However, this might be unrealistic. When the uncertainty of the model varies as a function of the input data, we refer to this as *heteroscedastic* (as opposed to *homoscedastic*, where the uncertainty is constant).

A simple way to model this is to train a neural network $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ that computes both the mean and the variance. For example, consider a shallow network with two outputs. We denote the first output as $\mathrm{f}_1[\mathbf{x}, \boldsymbol{\phi}]$ and use this to predict the mean, and we denote the second output as $\mathrm{f}_2[\mathbf{x}, \boldsymbol{\phi}]$ and use it to predict the variance.

There is one complication; the variance must be positive, but we can't guarantee that the network will always produce a positive output. To ensure that the computed variance is positive, we pass the second network output through a function that maps an arbitrary value to a positive one. A suitable choice is the squaring function, giving:

$$\begin{aligned} \mu &= \mathrm{f}_1[\mathbf{x}, \boldsymbol{\phi}] \\ \sigma^2 &= \mathrm{f}_2[\mathbf{x}, \boldsymbol{\phi}]^2, \end{aligned} \tag{5.14}$$

which results in the loss function:

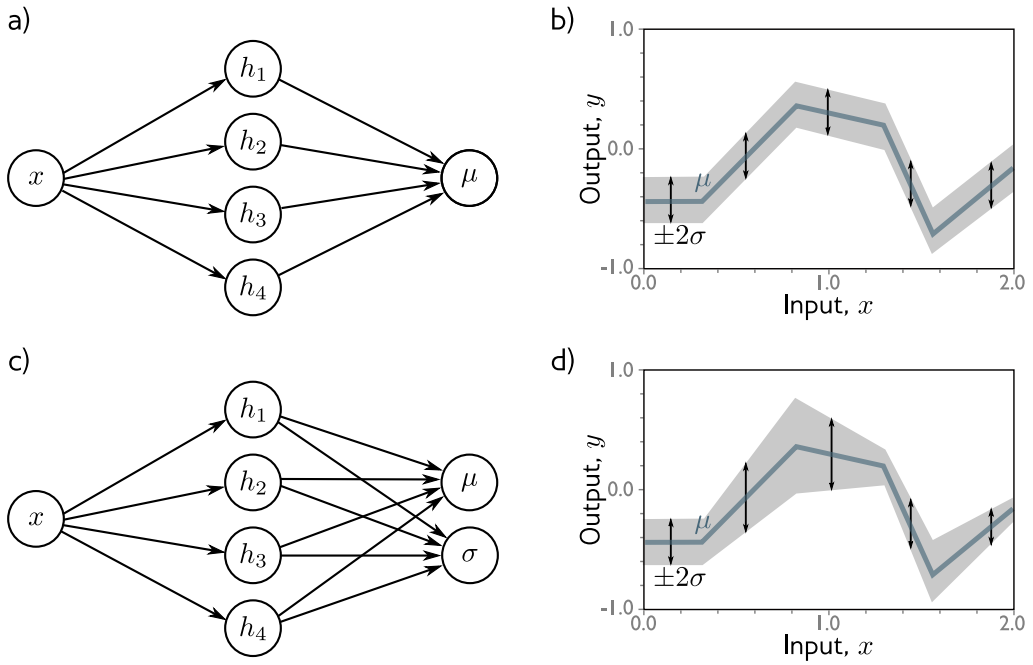

Figure 5.4 Homoscedastic vs. heteroscedastic regression. a) A shallow neural network for homoscedastic regression predicts just the mean μ of the output distribution from the input x. b) The result is that while the mean (blue line) is a piecewise linear function of the input x, the variance is constant everywhere (arrows and gray region show ± 2 standard deviations). c) A shallow neural network for heteroscedastic regression also predicts the variance σ^2 (or more precisely computes its square root, which we then square). d) The result is that the standard deviation also becomes a piecewise linear function of the input x.

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmin}} \left[-\sum_{i=1}^{I} \log \left[\frac{1}{\sqrt{2\pi \mathrm{f}_2[\mathbf{x}_i, \boldsymbol{\phi}]^2}} \right] - \frac{(y_i - \mathrm{f}_1[\mathbf{x}_i, \boldsymbol{\phi}])^2}{2\mathrm{f}_2[\mathbf{x}_i, \boldsymbol{\phi}]^2} \right]. \tag{5.15}$$

Homoscedastic and heteroscedastic models are compared in figure 5.4.

5.4 Example 2: multivariate regression

In multivariate regression, the goal is to predict a multidimensional target $\mathbf{y} \in \mathbb{R}^{D_o}$ from input data $\mathbf{x}$, where D_o is the number of target dimensions. Following the recipe in section 5.2, we first select a distribution over this domain. One possible choice would

be the multivariate normal distribution, which describes the covariance of the prediction errors. However, a simpler approach is just to assume that the errors in each dimension d are independent and use a product of univariate normal distributions:

$$Pr(\mathbf{y}|\boldsymbol{\mu}, \sigma^2) = \prod_{d=1}^{D_o} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(y_d - \mu_d)^2}{2\sigma^2}\right], \tag{5.16}$$

where y_d is the d^{th} entry of the ground truth output $\mathbf{y}$. Each distribution has a different mean μ_d (stored in a vector $\boldsymbol{\mu}$), but they all have the same variance σ^2.

We then use the machine learning model to predict some or all of the parameters of this distribution. Let's assume that we have a neural network model $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ with D_o outputs $\mathrm{f}_d[\mathbf{x}, \boldsymbol{\phi}]$ that predict the means μ_d. The likelihood is now:

$$Pr(\mathbf{y}|\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}], \sigma^2) = \prod_{d=1}^{D_o} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(y_d - \mathrm{f}_d[\mathbf{x}, \boldsymbol{\phi}])^2}{2\sigma^2}\right]. \tag{5.17}$$

The negative log-likelihood loss function for a training dataset $\{\mathbf{x}_i, \mathbf{y}_i\}$ is:

$$\begin{aligned}
\hat{\boldsymbol{\phi}} &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}} \left[-\sum_{i=1}^{I} \log\left[Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}], \sigma^2)\right]\right] \\
&= \underset{\boldsymbol{\phi}}{\operatorname{argmin}} \left[-\sum_{i=1}^{I} \log\left[\prod_{d=1}^{D_o} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(y_{id} - \mathrm{f}_d[\mathbf{x}_i, \boldsymbol{\phi}])^2}{2\sigma^2}\right]\right]\right] \\
&= \underset{\boldsymbol{\phi}}{\operatorname{argmin}} \left[\sum_{i=1}^{I}\sum_{d=1}^{D_o} (y_{id} - \mathrm{f}_d[\mathbf{x}_i, \boldsymbol{\phi}])^2\right],
\end{aligned} \tag{5.18}$$

where y_{id} is the d^{th} element of training target $\mathbf{y}_i$, and we have used similar steps as in equation 5.10 between the second and third lines. Again, this results in a least-squares formulation. For a new input $\mathbf{x}$, the prediction for $\mathbf{y}$ is the vector of means $\boldsymbol{\mu} = \mathbf{f}[\mathbf{x}, \hat{\boldsymbol{\phi}}]$.

5.4.1 Estimating variances

Problems 5.1-5.3

We may wish to estimate a vector $\mathbf{y}$ of quantities that are very different in magnitude. For example, consider predicting the height in meters and the weight in kilos of a human being from their age. We would expect the weight to have a much larger variance, simply because of the difference in units. If we use the above criterion, then the network will devote more effort to minimizing errors in the weight than to errors in the height.

One approach is to learn a separate variance σ_d^2 for each dimension as part of the loss function together with the network parameters. However, a commonly used practical alternative is just to scale the output dimensions to have the same empirical variance before training the model and then rescale the outputs of the model in the opposite way to make the final prediction.

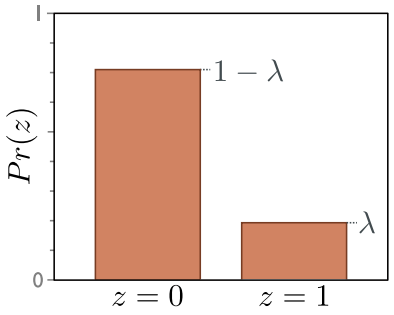

Figure 5.5 Bernoulli distribution. The Bernoulli distribution is defined on the domain $z \in \{0, 1\}$ and has a single parameter λ that denotes the probability of observing $z = 1$. It follows that the probability of observing $z = 0$ is $1 - \lambda$.

5.5 Example 3: binary classification

In *binary classification*, the goal is to predict which of two discrete classes $y \in \{0, 1\}$ the input data $\mathbf{x}$ belongs to. In this context, we refer to y as a *label*. Examples of binary classification include (i) predicting whether a restaurant review is positive ($y = 1$) or negative ($y = 0$) from text data $\mathbf{x}$ and (ii) predicting whether a tumor is present ($y = 1$) or absent ($y = 0$) from an MRI scan $\mathbf{x}$.

Once again, we follow the recipe from section 5.2 to construct the loss function. First, we choose a probability distribution over the output space $y \in \{0, 1\}$. A suitable choice is the Bernoulli distribution, which is defined on the domain $\{0, 1\}$. This has a single parameter $\lambda \in [0, 1]$ that represents the probability that y takes the value one (figure 5.5):

$$Pr(y|\lambda) = \begin{cases} 1-\lambda & y = 0 \\ \lambda & y = 1 \end{cases}, \tag{5.19}$$

which can equivalently be written as

$$Pr(y|\lambda) = (1-\lambda)^{1-y} \cdot \lambda^{y}. \tag{5.20}$$

Second, we set the machine learning model $\mathrm{f}[\mathbf{x}, \boldsymbol{\phi}]$ to predict the single distribution parameter λ. However, λ can only take values in the range $[0, 1]$, and we cannot guarantee that the network output will lie in this range. Consequently, we pass the network output through a function that maps the real line $\mathbb{R}$ to $[0, 1]$. A suitable function is the *logistic sigmoid* (figure 5.6):

Problem 5.4

$$\mathrm{sig}[z] = \frac{1}{1+\exp[-z]}. \tag{5.21}$$

Hence, we predict the distribution parameter as $\lambda = \mathrm{sig}[\mathrm{f}[\mathbf{x}, \boldsymbol{\phi}]]$. The likelihood is now:

$$Pr(y|\mathbf{x}) = (1 - \mathrm{sig}[\mathrm{f}[\mathbf{x}|\boldsymbol{\phi}]])^{1-y} \cdot \mathrm{sig}[\mathrm{f}[\mathbf{x}|\boldsymbol{\phi}]]^{y}. \tag{5.22}$$

This is depicted in figure 5.7 for a shallow neural network model. The loss function is the negative log-likelihood of the training set:

$$L[\boldsymbol{\phi}] = \sum_{i=1}^{I} -(1-y_i)\log\left[1 - \mathrm{sig}[\mathrm{f}[\mathbf{x}_i|\boldsymbol{\phi}]]\right] - y_i \log\left[\mathrm{sig}[\mathrm{f}[\mathbf{x}_i|\boldsymbol{\phi}]]\right]. \tag{5.23}$$

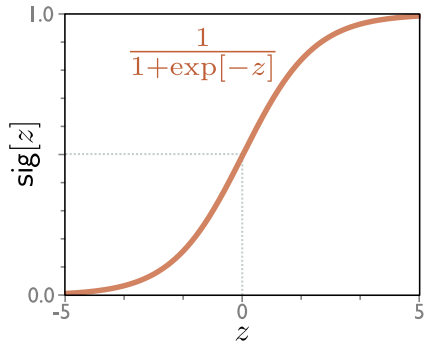

Figure 5.6 Logistic sigmoid function. This function maps the real line $z \in \mathbb{R}$ to numbers between zero and one, and so $\text{sig}[z] \in [0, 1]$. An input of 0 is mapped to 0.5. Negative inputs are mapped to numbers below 0.5 and positive inputs to numbers above 0.5.

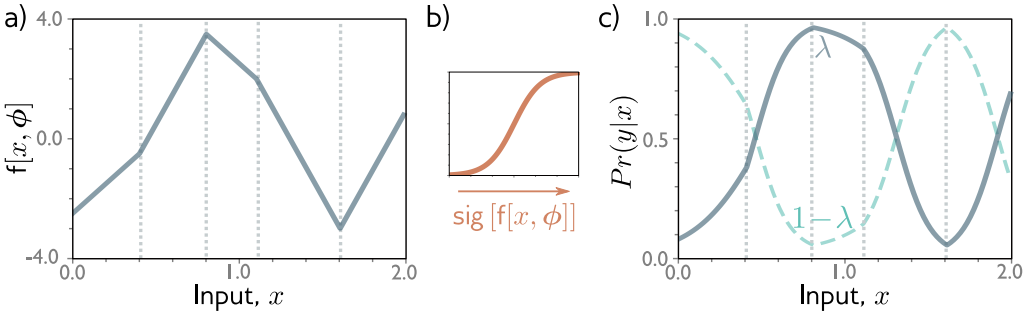

Figure 5.7 Binary classification model. a) The output of the network is a piecewise linear function that can take arbitrary real values. b) This is transformed by the logistic sigmoid function, which compresses these values to the range $[0, 1]$. c) The transformed output predicts the probability λ that $y = 1$ (solid line). The probability that $y = 0$ is hence $1 - \lambda$ (dashed line). For any fixed x (vertical slice) we retrieve the two values of a Bernoulli distribution similar to that in figure 5.5. The loss function favors model parameters that produce large values of λ at positions x_i that are associated with positive examples $y_i = 1$ and small values of λ at positions associated with negative examples $y_i = 0$.

For reasons to be explained in section 5.8, this is known as the *binary cross-entropy loss*.

The transformed model output $\text{sig}[\text{f}[\mathbf{x}, \boldsymbol{\phi}]]$ predicts the parameter λ of the Bernoulli distribution. This represents the probability that $y = 1$ and it follows that $1-\lambda$ represents the probability that $y = 0$. When we perform inference, we may want a point estimate of y, and so we set $y = 1$ if $\lambda > 0.5$ and $y = 0$ otherwise.

Problem 5.5

5.6 Example 4: multiclass classification

In multiclass classification, the goal is to classify an input data example $\mathbf{x}$ into $K > 2$

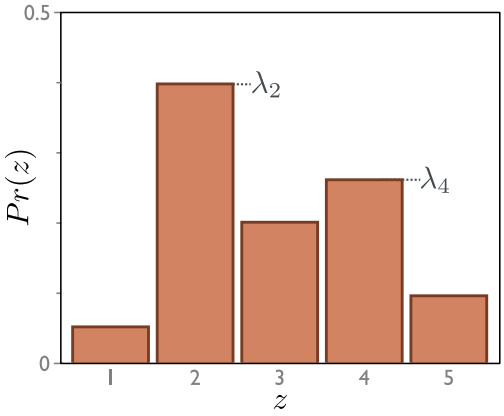

Figure 5.8 Categorical distribution. The categorical distribution assigns probabilities to $K > 2$ categories, with probabilities $\lambda_1, \lambda_2, \ldots, \lambda_K$. In this example, there are five categories, and so $K = 5$. To ensure that this is a valid probability distribution, each parameter λ_k must lie in the range $[0, 1]$ and all K parameters must sum to one.

classes so $y \in \{1, 2, \ldots, K\}$. Real-world examples include (i) predicting which of $K = 10$ digits y is present in an image $\mathbf{x}$ of a handwritten number, and (ii) predicting which of K possible words y follows an incomplete sentence $\mathbf{x}$.

We once more follow the recipe from section 5.2. We first choose a distribution over the prediction space y. In this case, we have $y \in \{1, 2, \ldots, K\}$, so we choose the categorical distribution (figure 5.8), which is defined on this domain. This has K parameters $\lambda_1, \lambda_2, \ldots, \lambda_K$, which determine the probability of each category:

$$Pr(y = k) = \lambda_k. \tag{5.24}$$

The parameters are constrained to take values between zero and one, and they must collectively sum to one to ensure a valid probability distribution.

Then we use a network $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ with K outputs to compute these K parameters from the input $\mathbf{x}$. Unfortunately, the network outputs will not necessarily obey the aforementioned constraints. Consequently, we pass the K outputs of the network through a function that ensures these constraints are respected. A suitable choice is the *softmax* function (figure 5.9). This takes an arbitrary vector of length K and returns a vector of the same length but where the elements are now in the range $[0, 1]$ and sum to one. The k^{th} output of the softmax function is:

$$\text{softmax}_k[\mathbf{z}] = \frac{\exp[z_k]}{\sum_{k=1}^{K} \exp[z_k]}, \tag{5.25}$$

where the exponential functions ensure positivity, and the sum in the denominator ensures that the K numbers sum to one.

The likelihood that input $\mathbf{x}$ has label y (figure 5.9) is hence:

$$Pr(y = k|\mathbf{x}) = \text{softmax}_k[\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]]. \tag{5.26}$$

The loss function is the negative log-likelihood of the training data:

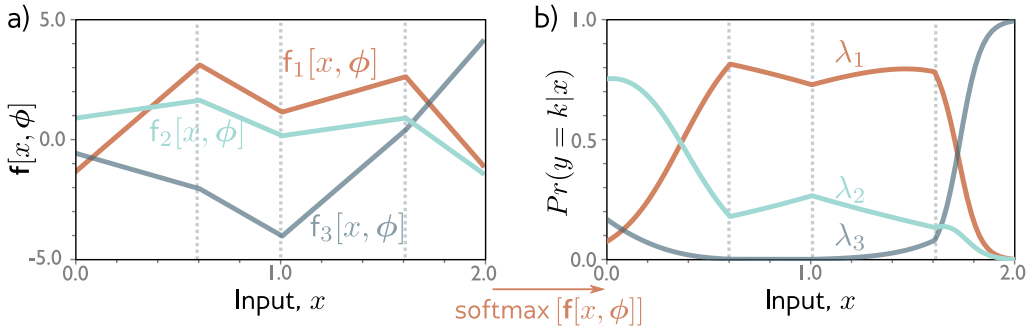

Figure 5.9 Multiclass classification for $K = 3$ classes. a) The network has three piecewise linear outputs, which can take arbitrary values. b) After the softmax function, these outputs are constrained to be non-negative and to sum to one. Hence, for a given input $\mathbf{x}$, we calculate a valid set of parameters of the categorical distribution. Any vertical slice of this plot produces three values that would form the heights of the bars in a categorical distribution similar to figure 5.8.

$$\begin{aligned} L[\boldsymbol{\phi}] &= -\sum_{i=1}^{I} \log\left[\text{softmax}_{y_i}\left[\mathbf{f}\left[\mathbf{x}_i, \boldsymbol{\phi}\right]\right]\right] \\ &= -\sum_{i=1}^{I} \mathrm{f}_{y_i}\left[\mathbf{x}_i, \boldsymbol{\phi}\right] - \log\left[\sum_{k=1}^{K} \exp\left[\,\mathrm{f}_k\left[\mathbf{x}_i, \boldsymbol{\phi}\right]\right]\right], \end{aligned} \tag{5.27}$$

where $\mathrm{f}_k[\mathbf{x}, \boldsymbol{\phi}]$ denotes the k^{th} output of the neural network. For reasons that will be explained in section 5.8, this is known as the *multiclass cross-entropy loss*.

For a given input $\mathbf{x}$, the transformed model output represents a categorical distribution over the possible predictions $y \in \{1, 2, \ldots, K\}$. To get a point estimate, we take the most probable category $\hat{y} = \text{argmax}_k[Pr(y = k|\mathbf{x})]$. This is whichever curve is highest in figure 5.9.

5.7 Predicting other data types

In this chapter, we have focused on regression and classification, because these problems are very common. However, to make different types of prediction, we simply choose an appropriate distribution over that domain and apply the recipe in section 5.2. Figure 5.10 enumerates a series of probability distributions and their prediction domains. Some of these are explored in the problems at the end of the chapter.

Problems 5.6–5.10

When we want to make two or more different types of prediction simultaneously, the usual approach is to assume that the errors in each are independent. For exam-

Data Type	Domain	Distribution	Use
univariate, continuous, unbounded	$y \in \mathbb{R}$	univariate normal	regression
univariate, continuous, unbounded	$y \in \mathbb{R}$	Laplace or t-distribution	robust regression
univariate, continuous, unbounded	$y \in \mathbb{R}$	mixture of Gaussians	multimodal regression
univariate, continuous, bounded below	$y \in \mathbb{R}^+$	exponential or gamma	predicting magnitude
univariate, continuous, bounded	$y \in [0, 1]$	beta	predicting proportions
multivariate, continuous, unbounded	$\mathbf{y} \in \mathbb{R}^K$	multivariate normal	multivariate regression
symmetric positive definite matrix	$\mathbf{Y} \in \mathbb{R}^{K \times K}$ $\mathbf{z}^T\mathbf{Y}\mathbf{z} > 0 \quad \forall \mathbf{z} \in \mathbb{R}^K$	Wishart	predicting covariances
univariate, continuous, circular	$y \in (-\pi, \pi]$	von Mises	predicting direction
univariate, discrete, binary	$y \in \{0, 1\}$	Bernoulli	binary classification
univariate, discrete, bounded	$y \in \{1, 2, \ldots, K\}$	categorical	multiclass classification
univariate, discrete, bounded below	$y \in [0, 1, 2, 3, \ldots]$	Poisson	predicting event counts
multivariate, discrete, permutation	$\mathbf{y} \in \text{Perm}[1, 2, \ldots K]$	Plackett–Luce	ranking

Figure 5.10 Distributions for loss functions for different prediction types.

ple, if we want to predict wind direction and strength, then we might choose the von Mises distribution (which is defined on circular domains) for the direction and the exponential distribution (which is defined on positive real numbers) for the strength. The independence assumption implies that the joint likelihood of the two predictions is just the product of individual likelihoods and that these terms will become additive when we compute the negative log-likelihood.

5.8 Cross-entropy loss

In this chapter, we developed loss functions based on minimizing negative log-likelihood. However, it is common in the literature to refer to *cross-entropy* losses. In this section, we introduce the cross-entropy loss and show that it is equivalent to using negative log-likelihood.

The cross-entropy loss is based on the idea of finding parameters $\boldsymbol{\theta}$ that minimize

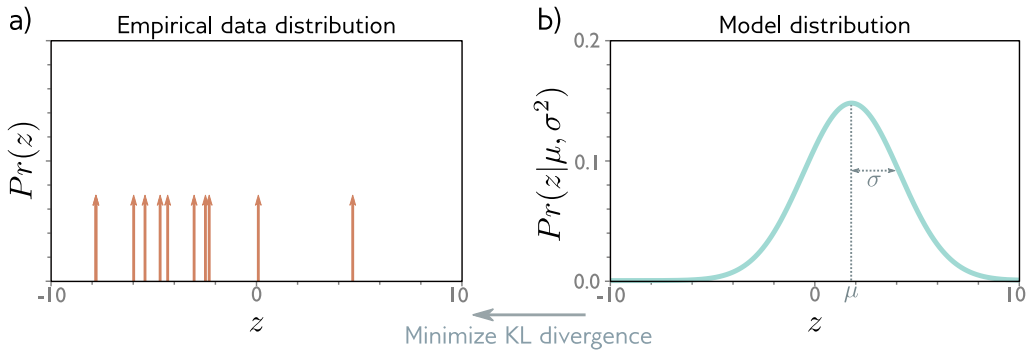

Figure 5.11 Cross-entropy method. a) Empirical distribution of training samples (arrows denote Dirac delta functions). b) Model distribution (a normal distribution with parameters $\boldsymbol{\theta} = \mu, \sigma^2$). In the cross-entropy approach, we minimize the distance (KL divergence) between these two distributions as a function of the model parameters $\boldsymbol{\theta}$.

the distance between the empirical distribution $q(y)$ of the observed data y and a model distribution $Pr(y|\boldsymbol{\theta})$ (figure 5.11). The "distance" between two probability distributions $q(z)$ and $p(z)$ can be evaluated using the Kullback–Leibler (KL) divergence:

Appendix B.1.6 KL Divergence

$$\text{KL}[q||p] = \int_{-\infty}^{\infty} q(z)\log[q(z)]dz - \int_{-\infty}^{\infty} q(z)\log[p(z)]dz. \tag{5.28}$$

Now consider that we observe an empirical distribution of data at points $\{y_i\}_{i=1}^{I}$. We can describe this as a weighted sum of point masses:

$$q(y) = \frac{1}{I}\sum_{i=1}^{I}\delta[y - y_i], \tag{5.29}$$

where $\delta[\bullet]$ is the Dirac delta function . We want to minimize the KL-divergence between the model distribution $Pr(y|\boldsymbol{\theta})$ and this empirical distribution:

Appendix B.1.8 Dirac delta

$$\begin{aligned}\hat{\boldsymbol{\theta}} &= \underset{\boldsymbol{\theta}}{\text{argmin}}\left[\int_{-\infty}^{\infty} q(y)\log[q(y)]dy - \int_{-\infty}^{\infty} q(y)\log\left[Pr(y|\boldsymbol{\theta})\right]dy\right] \\ &= \underset{\boldsymbol{\theta}}{\text{argmin}}\left[-\int_{-\infty}^{\infty} q(y)\log\left[Pr(y|\boldsymbol{\theta})\right]dy\right],\end{aligned} \tag{5.30}$$

where the first term disappears, as it has no dependence on $\boldsymbol{\theta}$. The remaining second term is known as the *cross-entropy*. It can be interpreted as the amount of uncertainty that remains in one distribution after taking into account what we already know from the other. Now, we substitute in the definition of $q(y)$ from equation 5.29:

$$\begin{aligned}\hat{\boldsymbol{\theta}} &= \underset{\theta}{\operatorname{argmin}}\left[-\int_{-\infty}^{\infty}\frac{1}{I}\sum_{i=1}^{I}\delta[y-y_i]\log\left[Pr(y|\boldsymbol{\theta})\right]dx\right] \\ &= \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\left[-\frac{1}{I}\sum_{i=1}^{I}\log\left[Pr(y_i|\boldsymbol{\theta})\right]\right] \\ &= \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\left[-\sum_{i=1}^{I}\log\left[Pr(y_i|\boldsymbol{\theta})\right]\right], \end{aligned} \tag{5.31}$$

where we have eliminated the constant scaling factor $1/I$ in the last line, as this does not affect the position of the minimum.

In machine learning, the distribution parameters $\boldsymbol{\theta}$ are computed by the model $\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}]$, and so we have:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[-\sum_{i=1}^{I}\log\left[Pr(y_i|\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}])\right]\right]. \tag{5.32}$$

This is exactly the negative log-likelihood criterion from the recipe in section 5.2.

It follows that the negative log-likelihood criterion (from maximizing the data likelihood) and the cross-entropy criterion (from minimizing the distance between the model and empirical data distributions) are equivalent.

5.9 Summary

We previously considered neural networks as directly predicting outputs $\mathbf{y}$ from data $\mathbf{x}$. In this chapter, we shifted perspective to think about neural networks as computing the parameters $\boldsymbol{\theta}$ of probability distributions $Pr(\mathbf{y}|\boldsymbol{\theta})$ over the output space. This led to a principled approach to building loss functions. We selected model parameters $\boldsymbol{\phi}$ that maximized the likelihood of the observed data under these distributions. We saw that this is equivalent to minimizing the negative log-likelihood.

The least squares criterion for regression is a natural consequence of this approach; it follows from the assumption that y is normally distributed and that we are predicting the mean. We also saw how the regression model can be (i) extended to estimate the uncertainty over the prediction, (ii) extended to make that uncertainty dependent on the input (the heteroscedastic model), and (iii) applied to multivariate predictions $\mathbf{y}$. We applied the same approach to both binary and multiclass classification and derived loss functions for each. We discussed how to tackle more complex data types. Finally, we argued that cross-entropy is an equivalent way to think about fitting models.

In previous chapters, we developed neural network models. In this chapter, we developed loss functions for deciding how well a model describes the training data for a given set of parameters. The next chapter considers model training, in which we aim to find the model parameters that minimize this loss.

Notes

Losses based on the normal distribution: Nix & Weigend (1994) and Williams (1996) investigated heteroscedastic nonlinear regression in which both the mean and the variance of the output are functions of the input. In the context of unsupervised learning, Burda et al. (2016) use a loss function based on a multivariate normal distribution with diagonal covariance, and Dorta et al. (2018) use a loss function based on a normal distribution with (approximated) full covariance.

Robust regression: Qi et al. (2020) investigate the properties of regression models that minimize mean absolute error rather than mean squared error. This loss function follows from assuming a Laplace distribution over the outputs and estimates the median output for a given input rather than the mean. Barron (2019) presents a loss function that parameterizes the degree of robustness. When interpreted in a probabilistic context, it yields a family of univariate probability distributions that includes the normal and Cauchy distributions as special cases.

Estimating quantiles: Sometimes we may not want to estimate the mean or median in a regression task but may instead want to predict a quantile. For example, this would be useful for risk models, where we might want to know that the true value will be less than the predicted value 90% of the time. This is known as *quantile regression* (Koenker & Hallock 2001). This could be done by fitting a heteroscedastic regression model and then estimating the quantile based on the predicted normal distribution. Alternatively, the quantiles can be estimated directly using *quantile loss* (also known as *pinball loss*). In practice, this minimizes the absolute deviations of the data from the model but weights the deviations in one direction more than the other. Recent work has investigated simultaneously predicting multiple quantiles to get an idea of the overall distribution shape (Rodrigues & Pereira 2018).

Class imbalance and focal loss: Lin et al. (2017) address the issue of data imbalance in classification problems. If the number of examples in some classes are much greater than in others, then the standard maximum likelihood/cross-entropy loss does not work well; the model may concentrate on becoming more confident about well-classified examples from the dominant classes and classify less well-represented classes poorly. Lin et al. (2017) introduce *focal loss*, which adds a single extra parameter that down-weights the effect of well-classified examples and improves performance in this setting.

Learning to rank: Cao et al. (2007), Xia et al. (2008), and Chen et al. (2009) all used the Plackett–Luce model in loss functions for learning to rank data. This is the *listwise* approach to learning to rank as the model ingests an entire list of objects to be ranked at once. Alternative approaches are the *pointwise* approach, in which a single object is ingested by the model, and the *pairwise* approach, where the model ingests pairs of objects. Different approaches for learning to rank are summarized in Chen et al. (2009).

Probabilistic models for other data types: Fan et al. (2020) use a loss based on a Beta distribution for predicting values between zero and one. Jacobs et al. (1991) and Bishop (1994) investigated machine learning models for multimodal data. These model the output as mixture of Gaussians that is conditional on the input and are called *mixture density networks*. Prokudin et al. (2018) investigated several models for predicting direction, including one based on the von Mises distribution. Fallah et al. (2009) constructed loss functions for prediction counts using the Poisson distribution. Ng et al. (2017) used loss functions based on the gamma distribution to predict duration.

Non-probabilistic approaches: It is not strictly necessary to adopt the probabilistic approach discussed in this chapter, but this has become the default in recent years. In fact, any loss function that aims to reduce the distance between the model output and the training outputs will suffice, and distance can be defined in any way that seems sensible. There are

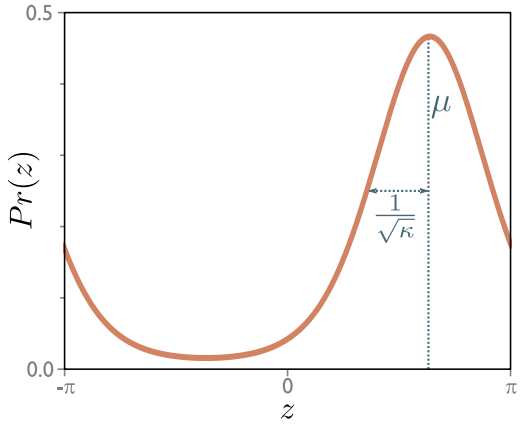

Figure 5.12 The von Mises distribution is defined over the circular domain $(-\pi, \pi]$. It has two parameters. The mean μ determines the position of the peak. The concentration $\kappa > 0$ acts like the inverse of the variance. Hence $1/\sqrt{\kappa}$ is roughly equivalent to the standard deviation in a normal distribution.

several well-known non-probabilistic machine learning models for classification, including support vector machines (Vapnik 1995; Cristianini & Shawe-Taylor 2000), which use *hinge loss*, and AdaBoost (Freund & Schapire 1995), which uses *exponential loss*.

Problems

Problem 5.1 Fill in the missing steps between lines two and three of equation 5.18.

Problem 5.2 Construct a loss function for making multivariate predictions $\mathbf{y}$ based on independent normal distributions as in section 5.4, but this time assume that we also want to estimate a separate variance σ_d^2 for each dimension. Assume these variances are constant, so the model is homoscedastic (i.e., the variances do not vary with the data).

Problem 5.3 Construct a loss function for making multivariate predictions $\mathbf{y}$ based on independent normal distributions with different variances for each dimension. However, this time assume that this is a heteroscedastic model so that both the means μ_d and variances σ_d^2 change as a function of the data.

Problem 5.4 Show that the logistic sigmoid function $\text{sig}[z]$ maps $z = -\infty$ to 0, $z = 0$ to 0.5 and $z = \infty$ to 1 where:

$$\text{sig}[z] = \frac{1}{1+\exp[-z]}. \tag{5.33}$$

Problem 5.5 The loss L for binary classification for a single training pair $\{\mathbf{x}, y\}$ is:

$$L = (1-y)\log\left[1-\text{sig}[\text{f}[\mathbf{x},\boldsymbol{\phi}]]\right] + y\log\left[\text{sig}[\text{f}[\mathbf{x},\boldsymbol{\phi}]]\right], \tag{5.34}$$

where $\text{sig}[\bullet]$ is defined in equation 5.33. Plot this loss as a function of the transformed network output $\text{sig}[\text{f}[\mathbf{x},\boldsymbol{\phi}]] \in [0,1]$ (i) when the training label $y = 0$ and (ii) when $y = 1$.

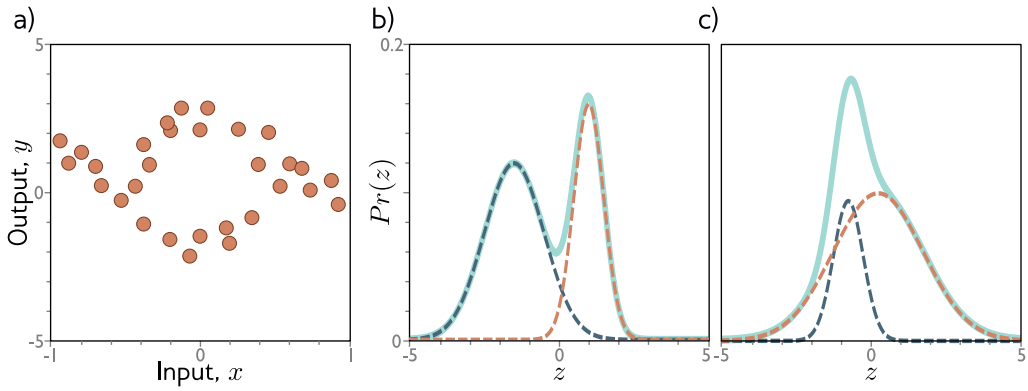

Figure 5.13 Multimodal data and mixture of Gaussians density. a) Example training data where, for intermediate values of the input x, the corresponding output y follows one of two paths. For example, at $x = 0$, the output y might be roughly -2 or $+3$ but is unlikely to be between these values. b) The mixture of Gaussians is a probability model that is suited to this kind of data. As the name suggests, the model is a weighted sum (solid cyan curve) of two or more normal distributions with different means and variances (here two weighted distributions, dashed blue and orange curves). When the means are far apart, this forms a multimodal distribution. c) When the means are close, the mixture can model unimodal, but non-normal densities.

Problem 5.6 Suppose we want to build a network that predicts the direction y in radians of the prevailing wind based on local measurements of barometric pressure $\mathbf{x}$. A suitable distribution over circular domains is the von Mises distribution (figure 5.12):

$$Pr(y|\mu,\kappa) = \frac{\exp[\kappa\cos[y-\mu]]}{2\pi\cdot\text{Bessel}_0[\kappa]}, \tag{5.35}$$

where μ is a measure of the mean direction and κ is a measure of the concentration (i.e., the inverse of the variance). The term $\text{Bessel}_0[\kappa]$ is a modified Bessel function of order 0.

Use the recipe from section 5.2 to develop a loss function for learning the parameter μ of a model $\text{f}[\mathbf{x},\boldsymbol{\phi}]$ to predict the most likely wind direction. Your solution should treat the concentration κ as constant. How would you perform inference?

Problem 5.7 Extend the model from problem 5.6 to predict both the wind direction and the wind speed and define the associated loss function.

Problem 5.8 Sometimes, the predictions y for a given input $\mathbf{x}$ are naturally multimodal as in figure 5.13a; there is more than one valid prediction for a given input. In this case, we might use a weighted sum of normal distributions as the distribution over the output. This is known as a *mixture of Gaussians* model . For example, a mixture of two Gaussians has distribution parameters $\boldsymbol{\theta} = \{\lambda, \mu_1, \sigma_1^2, \mu_2, \sigma_2^2\}$ is defined by:

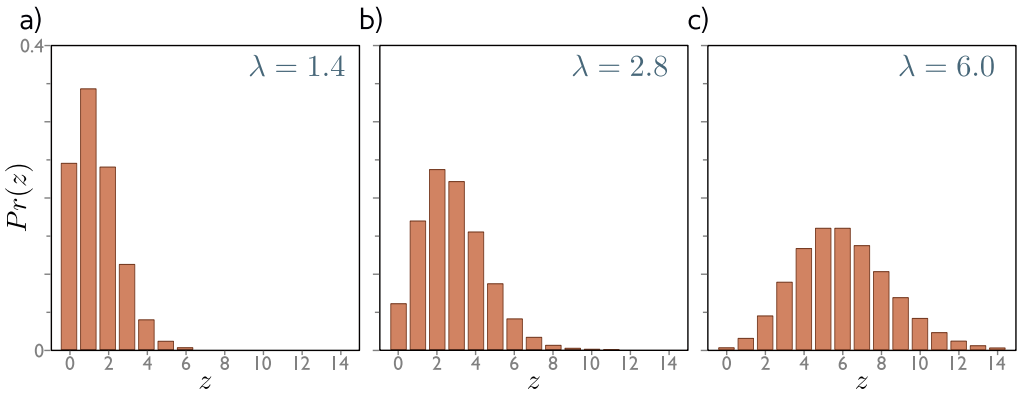

Figure 5.14 Poisson distribution. This discrete distribution is defined over non-negative integers $z \in \{0, 1, 2, \ldots\}$. It has a single parameter λ, which is known as the rate and is the mean of the distribution. a–c) Poisson distributions with rates of 1.4, 2.8, and 6.0, respectively.

$$Pr(y|\lambda, \mu_1, \mu_2, \sigma_1^2, \sigma_2^2) = \frac{\lambda}{\sqrt{2\pi\sigma_1^2}} \exp\left[\frac{-(y-\mu_1)^2}{2\sigma_1^2}\right] + \frac{1-\lambda}{\sqrt{2\pi\sigma_2^2}} \exp\left[\frac{-(y-\mu_2)^2}{2\sigma_2^2}\right], \tag{5.36}$$

where $\lambda \in [0, 1]$ controls the relative weight of the two normal distributions, which have means μ_1, μ_2 and variances σ_1^2, σ_2^2, respectively. This model can represent a distribution with two peaks (figure 5.13b) or a distribution with a single peak but a more complex shape (figure 5.13c).

Use the recipe from section 5.2 to construct a loss function for training a model $\mathbf{f}[x, \boldsymbol{\phi}]$ that takes input x, has parameters $\boldsymbol{\phi}$, and predicts a mixture of two Gaussians. The loss should be based on I training data pairs $\{x_i, y_i\}$. What possible problems do you foresee when we perform inference in this model?

Problem 5.9 Consider extending the model from problem 5.6 to predict the wind direction using a mixture of three von Mises distributions. Write an expression for the likelihood $Pr(y|\boldsymbol{\theta})$ for this model. How many outputs will the network need to produce?

Problem 5.10 Consider building a model to predict the number of pedestrians $y \in \{0, 1, 2, \ldots\}$ that will pass a given point in the city in the next minute, based on data $\mathbf{x}$ that contains information about the time of day, the longitude and latitude, and the type of neighborhood. A suitable distribution for modeling counts is the Poisson distribution (figure 5.14). This has a single parameter $\lambda > 0$ called the *rate* that represents the mean of the distribution. The distribution has probability density function:

$$Pr(y = k) = \frac{\lambda^k e^{-\lambda}}{k!}. \tag{5.37}$$

Use the recipe in section 5.2 to design a loss function for this model assuming that we have access to I training pairs $\{\mathbf{x}_i, y_i\}$.

Chapter 6

Fitting models

Chapters 3 and 4 described shallow and deep neural networks, respectively. These represent families of piecewise linear functions, where the particular function is determined by the parameters. Chapter 5 introduced the loss — a single number that represents the mismatch between the network predictions and the ground truth for a training set.

The loss depends on the parameters of the network, and we now consider how to find parameter values that minimize this loss. This is known as *learning* the parameters of the network, or simply as *training* or *fitting* the model. The process is to choose initial parameter values and then iterate the following two steps: (i) compute the derivatives (gradients) of the loss with respect to the parameters, and (ii) adjust the parameters based on the gradients to decrease the loss. After many steps, we hope to reach the overall minimum of the loss function.

This chapter tackles the second of these steps; we consider algorithms that adjust the parameters to decrease the loss. Chapter 7 discusses how to initialize the parameters and compute the gradients for neural networks.

6.1 Gradient descent

To fit a model, we need a training set $\{\mathbf{x}_i, \mathbf{y}_i\}$ of input/output pairs. We seek parameters $\boldsymbol{\phi}$ for the model $\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ that map the inputs $\mathbf{x}_i$ to the outputs $\mathbf{y}_i$ as well as possible. To this end, we define a loss function $\mathrm{L}[\boldsymbol{\phi}]$ that returns a single number that quantifies the mismatch in this mapping. The goal of an *optimization algorithm* is to find parameters $\hat{\boldsymbol{\phi}}$ that minimize the loss:

$$\hat{\boldsymbol{\phi}} = \operatorname{argmin}\left[\mathrm{L}[\boldsymbol{\phi}]\right]. \tag{6.1}$$

There are many families of optimization algorithms, but the standard methods for training neural networks are iterative. These algorithms initialize the parameters heuristically and then adjust them repeatedly in such a way that the loss decreases.

The simplest method in this class is *gradient descent*. This starts with initial parameters $\boldsymbol{\phi} = [\phi_0, \phi_1, \ldots, \phi_N]^T$ and iterates two steps:

Step 1. Compute the derivatives of the loss with respect to the parameters:

$$\frac{\partial L}{\partial \boldsymbol{\phi}} = \begin{bmatrix} \frac{\partial L}{\partial \phi_0} \\ \frac{\partial L}{\partial \phi_1} \\ \vdots \\ \frac{\partial L}{\partial \phi_N} \end{bmatrix}. \tag{6.2}$$

Step 2. Update the parameters according to the rule:

$$\boldsymbol{\phi} \longleftarrow \boldsymbol{\phi} - \alpha \frac{\partial L}{\partial \boldsymbol{\phi}}, \tag{6.3}$$

where the positive scalar α determines the magnitude of the change.

The first step computes the gradient of the loss function at the current position. This determines the uphill direction on the loss function. The second step moves a small distance α *downhill* (hence the negative sign). The parameter α may be fixed (in which case, we call it a *learning rate*), or we may perform a *line search* where we try several values of α to find the one that decreases the loss the most.

At the minimum of the loss function, the surface must be flat (or we could improve further by going downhill). Hence, the gradient will be zero and the parameters will stop changing. In practice, we monitor the gradient magnitude, and when it becomes too small, we terminate the algorithm.

6.1.1 Linear regression example

Consider applying gradient descent to the 1D linear regression model from chapter 2. This only has two parameters and so is easy to visualize. The model $\mathrm{f}[x, \boldsymbol{\phi}]$ maps a scalar input x to a scalar output y and has parameters $\boldsymbol{\phi} = [\phi_0, \phi_1]^T$ which represent the y-intercept and the slope:

$$\begin{aligned} y &= \mathrm{f}[x, \boldsymbol{\phi}] \\ &= \phi_0 + \phi_1 x. \end{aligned} \tag{6.4}$$

Given a dataset $\{x_i, y_i\}$ containing I input/output pairs, we choose the least squares loss function:

$$\begin{aligned} L[\boldsymbol{\phi}] = \sum_{i=1}^{I} \mathrm{l}_i &= \sum_{i=1}^{I} \left(\mathrm{f}[x_i, \boldsymbol{\phi}] - y_i\right)^2 \\ &= \sum_{i=1}^{I} \left(\phi_0 + \phi_1 x_i - y_i\right)^2 \end{aligned} \tag{6.5}$$

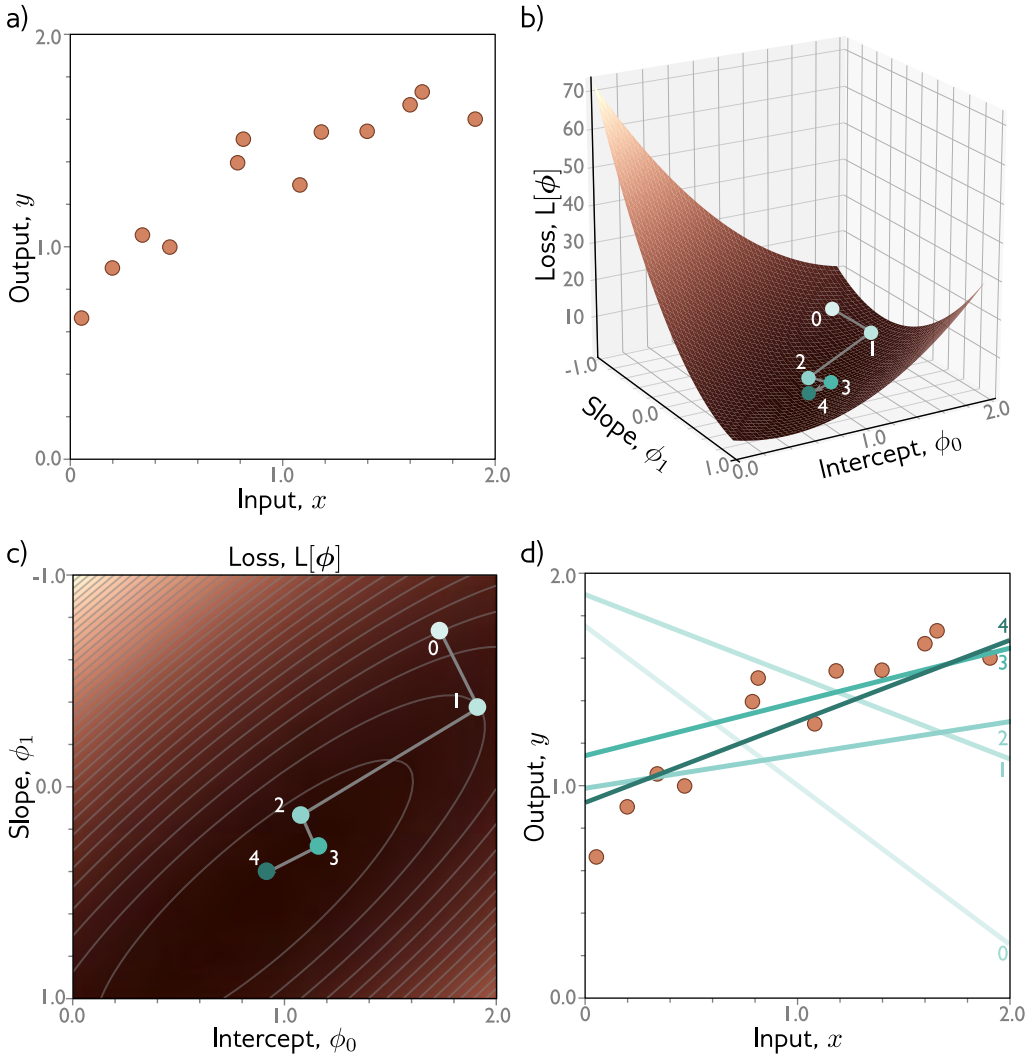

Figure 6.1 Gradient descent for the linear regression model. a) Training set of $I = 12$ input/output pairs $\{x_i, y_i\}$. b) Loss function showing iterations of gradient descent. We start at point 0 and move in the steepest downhill direction until we can improve no further to arrive at point 1. We then repeat this procedure. We measure the gradient at point 1 and move downhill to point 2 and so on. c) This can be visualized better as a heatmap, where the brightness represents the loss. After only four iterations, we are already close to the minimum. d) The model with the parameters at point 0 (lightest line) describes the data very badly, but each successive iteration improves the fit. The model with the parameters at point 4 (darkest line) is already a reasonable description of the training data.

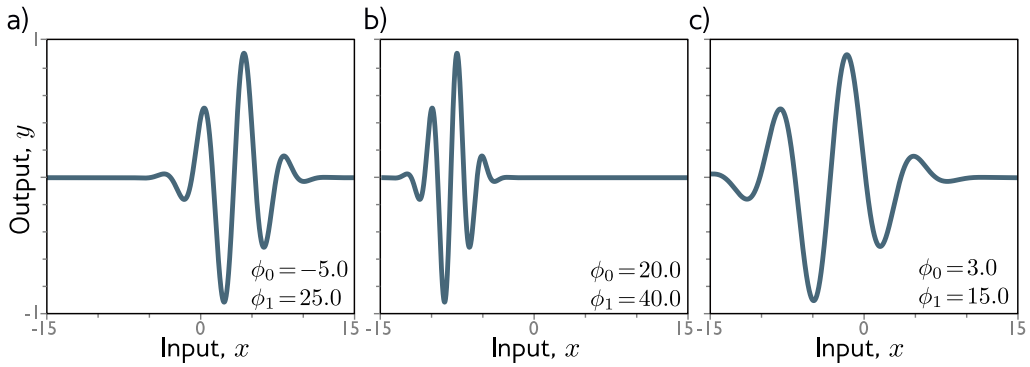

Figure 6.2 Gabor model. This nonlinear model maps scalar input x to scalar output y and has two parameters $\boldsymbol{\phi} = [\phi_0, \phi_1]^T$. The model represents a sinusoidal function that decreases in amplitude with distance from the center. The parameter $\phi_0 \in \mathbb{R}$ determines the position of the center. As ϕ_0 increases, the function moves leftwards. The parameter $\phi_1 \in \mathbb{R}^+$ stretches or squeezes the function along the x-axis relative to the center. As ϕ_1 increases, the function narrows. a–c) Three examples of the model with different parameters.

where the term $l_i = (\phi_0 + \phi_1 x_i - y_i)^2$ is the individual contribution to the loss from the i^{th} training example.

The derivative of the loss function with respect to the parameters can be decomposed into the sum of the derivatives of the individual contributions:

$$\frac{\partial L}{\partial \boldsymbol{\phi}} = \frac{\partial}{\partial \boldsymbol{\phi}} \sum_{i=1}^{I} l_i = \sum_{i=1}^{I} \frac{\partial l_i}{\partial \boldsymbol{\phi}}, \tag{6.6}$$

Problem 6.1

where these are given by:

$$\frac{\partial l_i}{\partial \boldsymbol{\phi}} = \begin{bmatrix} \frac{\partial l_i}{\partial \phi_0} \\ \frac{\partial l_i}{\partial \phi_1} \end{bmatrix} = \begin{bmatrix} 2(\phi_0 + \phi_1 x_i - y_i) \\ 2x_i(\phi_0 + \phi_1 x_i - y_i) \end{bmatrix}. \tag{6.7}$$

Figure 6.1 shows the progression of this algorithm as we iteratively compute the derivatives according to equations 6.6 and 6.7 and then update the parameters using the rule in equation 6.3. In this case, we have used a line search procedure to find the value of α that decreases the loss the most at each iteration.

6.1.2 Gabor model example

Loss functions for linear regression problems (figure 6.1c) always have a single well-defined global minimum. More formally, they are *convex*, which means that no chord

Problem 6.2

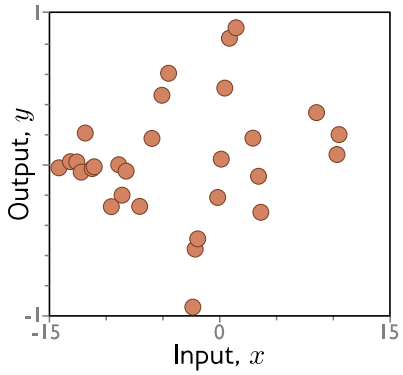

Figure 6.3 Training data for fitting the Gabor model. The training dataset consists of 28 input/output examples $\{x_i, y_i\}$. These were created by uniformly sampling $x_i \in [-15, 15]$, passing the samples through a Gabor model with parameters $\boldsymbol{\phi} = [0.0, 16.6]^T$, and adding normally distributed noise.

(line segment between two points on the surface) intersects the function. Convexity implies that wherever we initialize the parameters, we are bound to reach the minimum if we keep walking downhill; the training procedure really can't fail.

Unfortunately, loss functions for most nonlinear models, including both shallow and deep networks, are *nonconvex*. Visualization of neural network loss functions is challenging due to the number of parameters. Consequently, we'll first explore a simpler nonlinear model with two parameters to gain insight into the properties of nonconvex loss functions:

$$\mathrm{f}[x, \boldsymbol{\phi}] = \sin[\phi_0 + 0.06 \cdot \phi_1 x] \cdot \exp\left(-\frac{(\phi_0 + 0.06 \cdot \phi_1 x)^2}{8.0}\right). \tag{6.8}$$

This *Gabor model* maps scalar input x to scalar output y and consists of a sinusoidal component (creating an oscillatory function) multiplied by a negative exponential component (causing the amplitude to decrease as we move from the center). It has two parameters $\boldsymbol{\phi} = [\phi_0, \phi_1]^T$, where $\phi_0 \in \mathbb{R}$ determines the mean position of the function and $\phi_1 \in \mathbb{R}^+$ stretches or squeezes it along the x-axis (figure 6.2).

Problems 6.3–6.6

Consider a training set of I examples $\{x_i, y_i\}$ (figure 6.3). The least squares loss function for I training examples is defined as:

$$L[\boldsymbol{\phi}] = \sum_{i=1}^{I} \left(\mathrm{f}[x_i, \boldsymbol{\phi}] - y_i\right)^2. \tag{6.9}$$

Once more, the goal is to find the parameters $\hat{\boldsymbol{\phi}}$ that minimize this loss.

6.1.3 Local minima and saddle points

The loss function associated with the Gabor model for this dataset is depicted in figure 6.4. There are numerous *local minima* (cyan circles). Here the gradient is zero and the loss increases if we move in any direction, but we are *not* at the overall minimum of the function. If we start in a random position and use gradient descent to go downhill, there is no guarantee that we will wind up at the global minimum (green point) and find

Problem 6.4

Problem 6.7

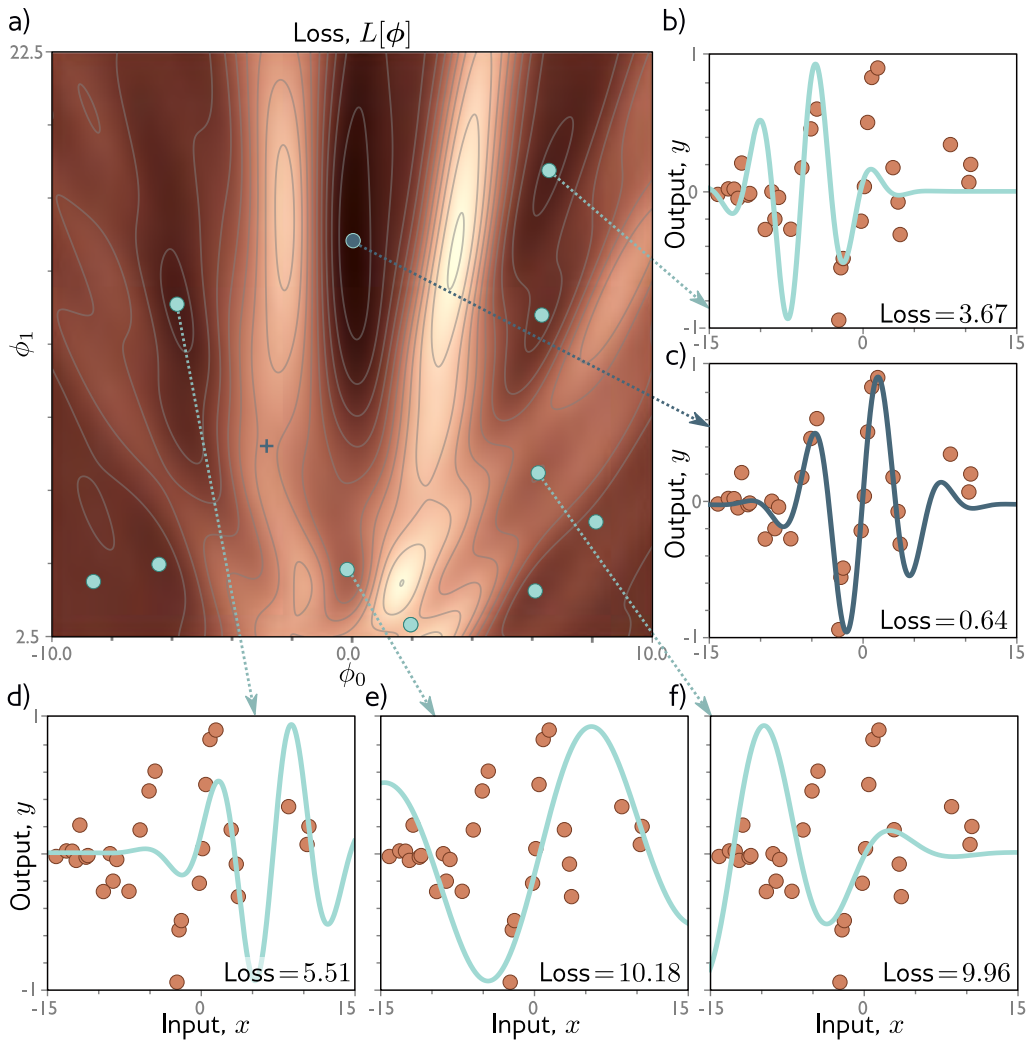

Figure 6.4 Loss function for the Gabor model. a) The loss function is nonconvex, with multiple local minima (cyan points) in addition to the global minimum (green point). It also contains saddle points where the gradient is locally zero, but the function is increasing in one direction and decreasing in the other. The green cross is an example of a saddle point; the function decreases as we move horizontally in either direction but increases as we move vertically. b–f) Models associated with the different minima. In each case, there is no small change that decreases the loss. Panel (c) shows the best model, which has a loss of 0.64.

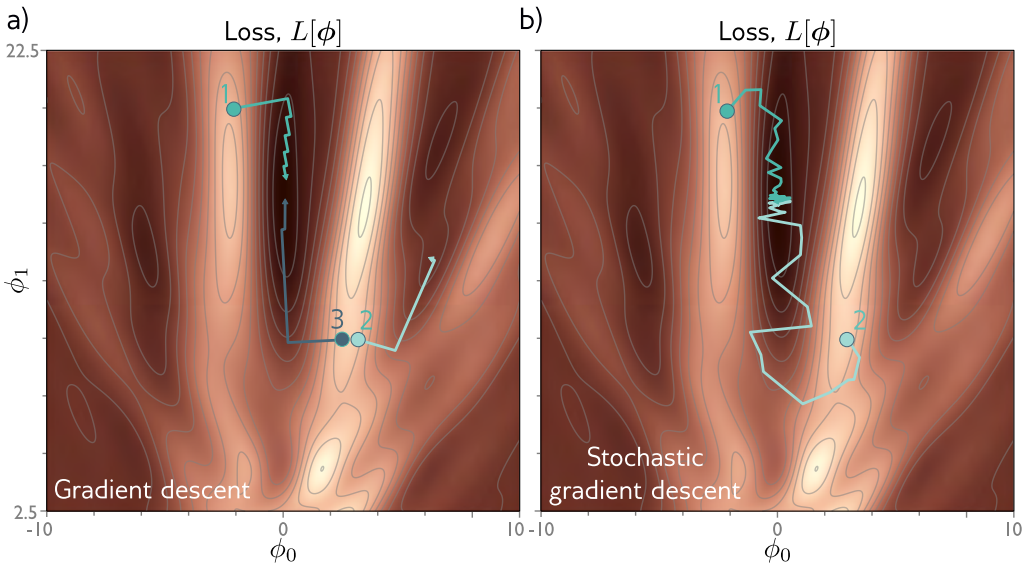

Figure 6.5 Gradient descent vs. stochastic gradient descent. a) Gradient descent. As long as the gradient descent algorithm is initialized in the right "valley" of the loss function (e.g., points 1 and 3), the parameter estimate will move steadily toward the global minimum. However, if it is initialized outside this valley (e.g., point 2) then it will descend toward one of the local minima. b) Stochastic gradient descent adds noise to the optimization process, and so it is possible to escape from starting points that are not in the right valley (e.g., point 2) and still reach the global minimum.

the best parameters (figure 6.5a). It's equally or even more likely that the algorithm will terminate in one of the local minima. Furthermore, there is no way of knowing whether there is a better solution elsewhere.

In addition, the loss function contains *saddle points* (e.g., the blue cross in figure 6.4). Here, the gradient is zero, but the function is increasing in some directions and decreasing in others. If the current position does not lie exactly on the saddle point, then gradient descent algorithms can escape by moving downhill. However, the surface near the saddle point is very flat, and so it's hard to be sure that training has not converged.

6.2 Stochastic gradient descent

The Gabor model has two parameters, and so we could find the global minimum by either (i) exhaustively searching the parameter space, or (ii) repeatedly starting gradient descent from different positions and choosing the result with the lowest loss. However,

neural network models can have millions of parameters, and so here neither approach is practical. In short, using gradient descent to find the global optimum of a high-dimensional loss function is challenging. We can find *a* minimum, but there is no way to tell whether this is the global minimum or even a good one.

One of the main problems is that the final destination of a gradient descent algorithm is entirely determined by the starting point. *Stochastic gradient descent (SGD)* attempts to remedy this problem by adding some noise to the gradient at each step. The solution still moves downhill on average but at any given iteration the direction chosen is not necessarily in the steepest downhill direction. Indeed, it might not be downhill at all. The SGD algorithm has the possibility of moving temporarily uphill and hence moving from one "valley" of the loss function to another (figure 6.5b).

6.2.1 Batches and epochs

The mechanism for introducing randomness is simple. At each iteration, the algorithm chooses a random subset of the training data and computes the gradient from these examples alone. This subset is known as a *minibatch* or *batch* for short. The update rule for the model parameters $\boldsymbol{\phi}_t$ at iteration t is hence:

$$\boldsymbol{\phi}_{t+1} \longleftarrow \boldsymbol{\phi}_t - \alpha \sum_{i \in \mathcal{B}_t} \frac{\partial l_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}, \tag{6.10}$$

where $\mathcal{B}_t$ is a set containing the indices of the input/output pairs in the current batch and, as before, l_i is the loss due to the i^{th} example. The term α is the learning rate, and together with the gradient magnitude, this determines the distance moved at each iteration. The learning rate is chosen at the start of the procedure and does not depend on the local properties of the function.

The batches are usually drawn from the dataset without replacement. The algorithm works through the training examples until it has used all the data, at which point it starts sampling from the full training dataset again. A single pass through all the data is referred to as an *epoch*. A batch may be as small as a single example, or as large the entire dataset. The latter case is referred to as *full-batch gradient descent* and is identical to regular (nonstochastic) gradient descent.

An alternative interpretation of SGD is that it computes the gradient of a different loss function at each iteration; the loss function depends on both the model and the training data, and so will be different for each randomly selected batch. In this view, SGD performs deterministic gradient descent on a constantly changing loss function (figure 6.6). However, despite this variability, the expected loss and expected gradients at any point remain the same as for gradient descent.

6.2.2 Properties of stochastic gradient descent

Stochastic gradient descent has several attractive features. First, although it adds noise to the trajectory, it still improves the fit to a subset of the data at each iteration.

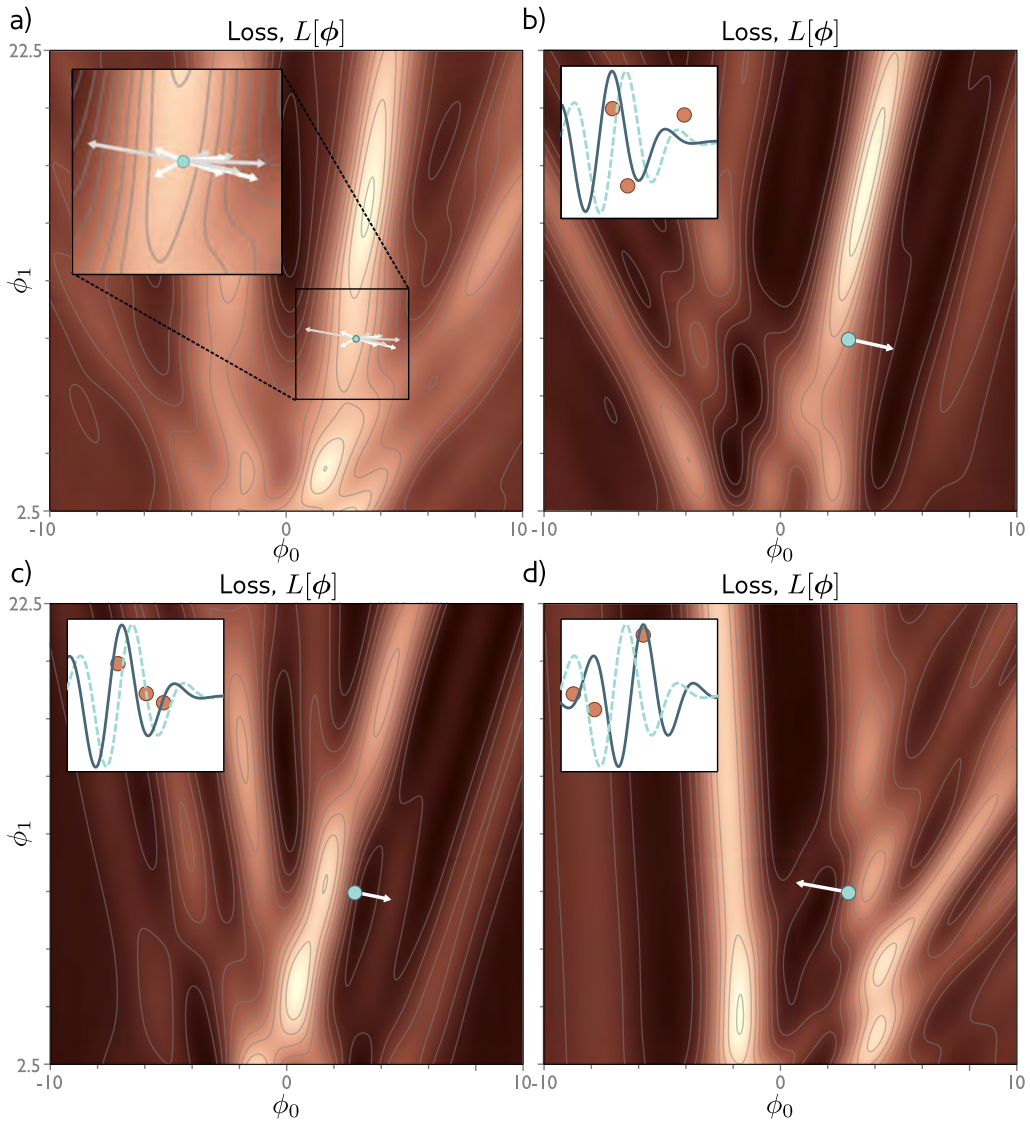

Figure 6.6 Alternative view of SGD for the Gabor model with a batch size of three. a) Loss function for the entire training dataset. At each iteration, there is a probability distribution of possible parameter changes (inset shows samples). These correspond to different choices of the three batch elements. b) Loss function for one possible batch. The SGD algorithm moves in the downhill direction on this function for a distance that is determined by the learning rate and the local gradient magnitude. The current model (dashed function in inset) changes to better fit the batch data (solid function). c) A different batch creates a different loss function and results in a different update. d) For this batch, the algorithm moves downhill with respect to the batch loss function but *uphill* with respect to the global loss function in panel (a). This is how SGD can escape local minima.

Hence, the updates tend to be sensible even if they are not optimal. Second, because it draws training examples without replacement and iterates through the dataset, the training examples still all contribute equally. Third, it is less computationally expensive to compute the gradient from just a subset of the training data. Finally, there is some evidence that SGD finds parameters for neural networks that cause them to generalize well to new data in practice (see section 9.2).

Stochastic gradient descent does not necessarily "converge" in the traditional sense. However, the hope is that when we are close to the global minimum, all the data points are described well by the model. Consequently, the gradient will be small whichever batch is chosen, and the parameters will cease to change much. In practice, stochastic gradient descent is often applied with a *learning rate schedule*. The learning rate α starts at a relatively high value and is decreased by a constant factor every N epochs. The logic is that in the early stages of training, we want the algorithm to explore the parameter space, jumping from valley to valley to find a sensible region. In later stages, we are in roughly the right place and are more concerned with fine-tuning the parameters, and so we make smaller changes.

6.3 Momentum

A common modification to stochastic gradient descent is to add a *momentum* term. We update the parameters with a weighted combination of the gradient computed from the current batch and the direction moved in the previous step:

$$\begin{aligned} \mathbf{m}_{t+1} &\leftarrow \beta \cdot \mathbf{m}_t + (1-\beta) \sum_{i \in \mathcal{B}_t} \frac{\partial l_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}} \\ \boldsymbol{\phi}_{t+1} &\leftarrow \boldsymbol{\phi}_t - \alpha \cdot \mathbf{m}_{t+1}, \end{aligned} \tag{6.11}$$

where $\mathbf{m}_t$ is the momentum (which drives the update at iteration t), $\beta \in [0, 1)$ controls the degree to which the gradient is smoothed over time, and α is the learning rate.

The recursive formulation of the momentum calculation means that the gradient step is an infinite weighted sum of all the previous gradients, where the weights get smaller as we move back in time. The effective learning rate is increased if all these gradients are aligned over multiple iterations but decreased if the gradient direction repeatedly changes as the terms in the sum cancel out. The overall effect is a smoother trajectory and reduced oscillatory behavior in valleys (figure 6.7).

Problem 6.8

6.3.1 Nesterov accelerated momentum

The momentum term can be thought of as a coarse prediction of where the SGD algorithm will move next. Nesterov accelerated momentum (figure 6.8) computes the gradients at this predicted point rather than at the current point:

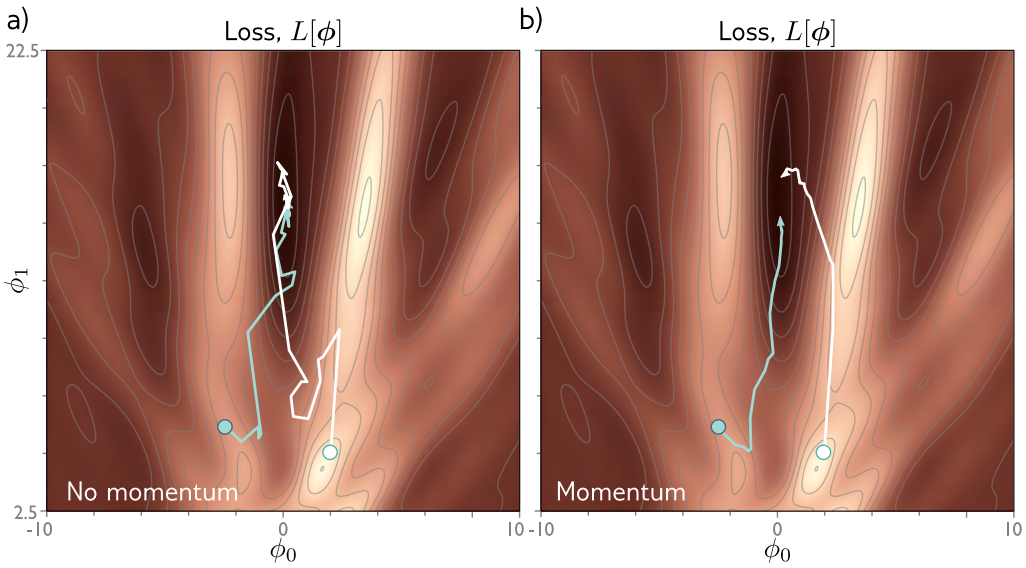

Figure 6.7 Stochastic gradient descent with momentum. a) Regular stochastic descent takes a very indirect path toward the minimum. b) With a momentum term, the change at the current step is a weighted combination of the previous change and the gradient computed from the batch. This smooths out the trajectory and increases the speed of convergence.

$$\begin{aligned} \mathbf{m}_{t+1} &\leftarrow \beta \cdot \mathbf{m}_t + (1-\beta) \sum_{i \in \mathcal{B}_t} \frac{\partial l_i[\boldsymbol{\phi}_t - \alpha \cdot \mathbf{m}_t]}{\partial \boldsymbol{\phi}} \\ \boldsymbol{\phi}_{t+1} &\leftarrow \boldsymbol{\phi}_t - \alpha \cdot \mathbf{m}_{t+1}, \end{aligned} \tag{6.12}$$

where now the gradients are evaluated at $\boldsymbol{\phi}_t - \alpha \cdot \mathbf{m}_t$. One way to think about this is that the gradient term now corrects the path provided by momentum alone.

6.4 Adam

Gradient descent with a fixed step size has the following undesirable property: it makes large adjustments to parameters associated with large gradients (where perhaps we should be more cautious), and small adjustments to parameters associated with small gradients (where perhaps we should explore further). This means that when the gradient of the loss surface is much steeper in one direction than another, it is difficult to

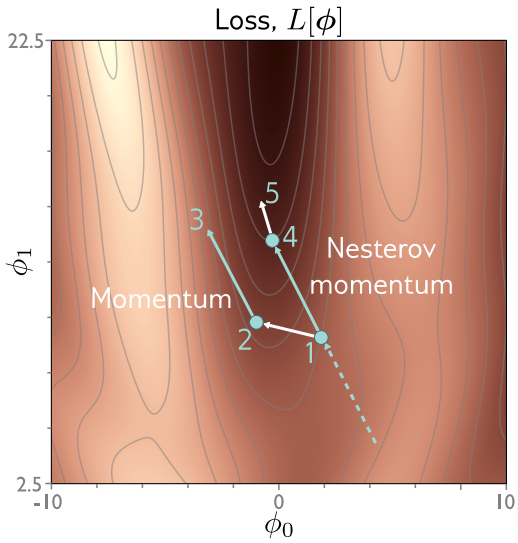

Figure 6.8 Nesterov accelerated momentum. The solution has just traveled along the dashed line to arrive at point 1. A traditional momentum update measures the gradient at point 1, moves some distance in this direction to point 2, and then adds the momentum term from the previous iteration (i.e., in the same direction as the dashed line), arriving at point 3. The Nesterov momentum update first applies the momentum term (moving from point 1 to point 4), and then measures the gradient and applies an update to arrive at point 5.

choose a learning rate that (i) makes good progress in both directions and (ii) is stable (figures 6.9a–b).

A very simple approach is to normalize the gradients so that we move a fixed distance governed by the learning rate in each direction. To do this, we first measure the gradient $\mathbf{m}_{t+1}$ and the pointwise squared gradient $\mathbf{v}_{t+1}$:

$$\begin{aligned} \mathbf{m}_{t+1} &\leftarrow \frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}} \\ \mathbf{v}_{t+1} &\leftarrow \frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}^2 . \end{aligned} \tag{6.13}$$

Then we apply the update rule:

$$\boldsymbol{\phi}_{t+1} \leftarrow \boldsymbol{\phi}_t - \alpha \cdot \frac{\mathbf{m}_{t+1}}{\sqrt{\mathbf{v}_{t+1}} + \epsilon}, \tag{6.14}$$

where the square root and division are both pointwise, α is the learning rate, and ϵ is a small constant that prevents division by zero when the gradient magnitude is zero. The term $\mathbf{v}_{t+1}$ is the squared gradient, and the positive root of this is used to normalize the gradient itself so all that remains is the sign in each coordinate direction. The result is that the algorithm moves a fixed distance α along each coordinate, where the direction is determined by whichever way is downhill (figure 6.9c). This simple algorithm makes good progress in both directions but will not converge unless it happens to land exactly at the minimum. Instead, it will bounce back and forth around the minimum.

Adaptive moment estimation or *Adam* takes this idea and adds momentum to both the estimate of the gradient and the squared gradient:

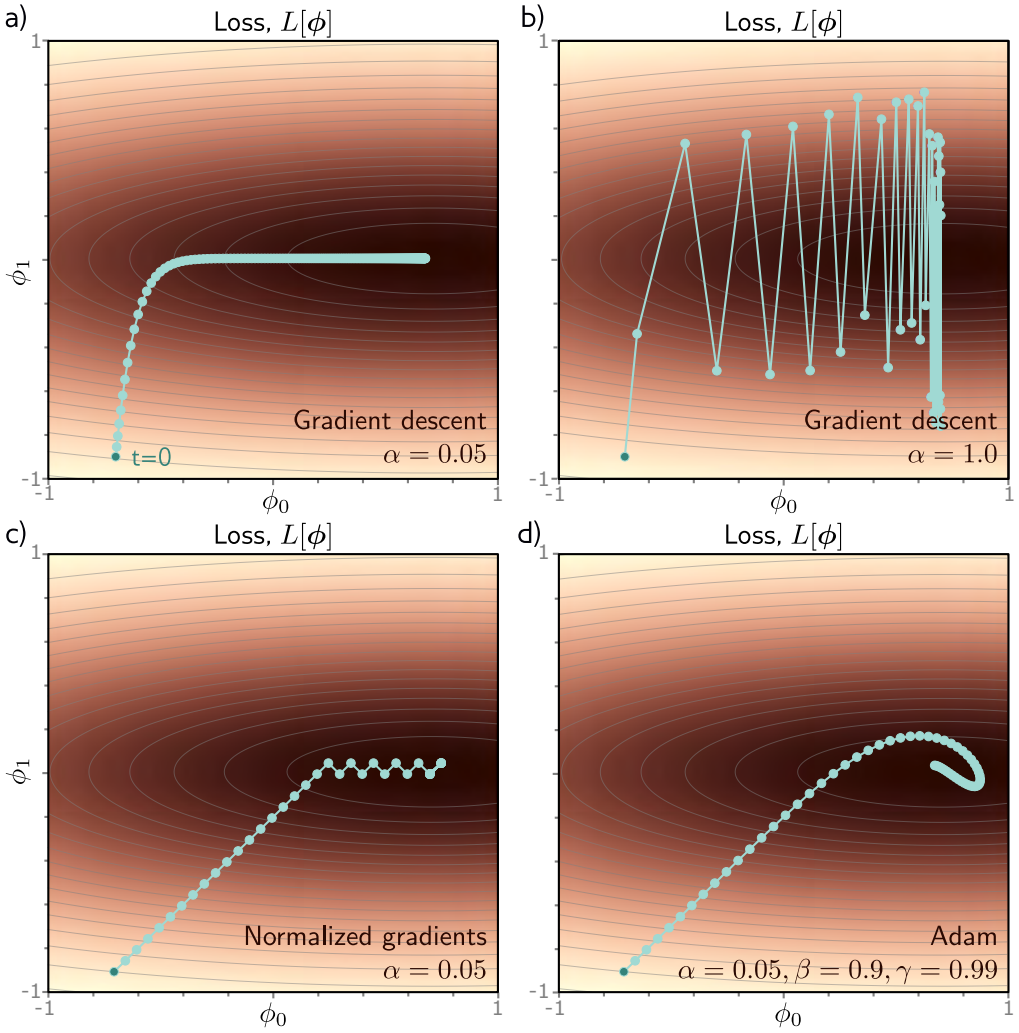

Figure 6.9 Adaptive moment estimation (Adam). a) This loss function changes quickly in the vertical direction, but slowly in the horizontal direction. If we run full-batch gradient descent with a learning rate that makes good progress in the vertical direction, then the algorithm takes a long time to reach the final horizontal position. b) If the learning rate is chosen so that the algorithm makes good progress in the horizontal direction, then it overshoots in the vertical direction and becomes unstable. c) A very simple approach is just to move a fixed distance along each axis at each step in such a way that we move downhill in both directions. This is accomplished by normalizing the gradient magnitude and retaining only the sign. However, this does not usually converge to the exact minimum but instead oscillates back and forth around it (here between the last two points). d) The Adam algorithm uses momentum in both the estimated gradient and the normalization term, which together create a smoother path.

$$\begin{aligned}\mathbf{m}_{t+1} &\leftarrow \beta \cdot \mathbf{m}_t + (1-\beta)\frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}} \\ \mathbf{v}_{t+1} &\leftarrow \gamma \cdot \mathbf{v}_t + (1-\gamma)\left(\frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}\right)^2,\end{aligned} \tag{6.15}$$

where β and γ are the momentum coefficients for the two statistics.

Using momentum is equivalent to taking a weighted average over the history of each of these statistics. At the start of the procedure, all the previous measurements are effectively zero, and so this results in unrealistically small estimates. Consequently, we modify these statistics using the rule:

$$\begin{aligned}\tilde{\mathbf{m}}_{t+1} &\leftarrow \frac{\mathbf{m}_{t+1}}{1-\beta^{t+1}} \\ \tilde{\mathbf{v}}_{t+1} &\leftarrow \frac{\mathbf{v}_{t+1}}{1-\gamma^{t+1}}.\end{aligned} \tag{6.16}$$

Since β and γ are in the range $[0, 1)$, the terms with exponents $t+1$ become smaller with each time-step, and so the denominator becomes closer to one and this modification has a diminishing effect.

Finally, we update the parameters as before, but with the modified terms:

$$\boldsymbol{\phi}_{t+1} \leftarrow \boldsymbol{\phi}_t - \alpha \cdot \frac{\tilde{\mathbf{m}}_{t+1}}{\sqrt{\tilde{\mathbf{v}}_{t+1}} + \epsilon}. \tag{6.17}$$

The result is an algorithm that can converge to the overall minimum and makes good progress in every direction in parameter space. Note that Adam is usually used in a stochastic setting where the gradients and their squares are computed from mini-batches:

$$\begin{aligned}\mathbf{m}_{t+1} &\leftarrow \beta \cdot \mathbf{m}_t + (1-\beta)\sum_{i \in \mathcal{B}_t} \frac{\partial l_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}} \\ \mathbf{v}_{t+1} &\leftarrow \gamma \cdot \mathbf{v}_t + (1-\gamma)\sum_{i \in \mathcal{B}_t} \left(\frac{\partial l_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}\right)^2,\end{aligned} \tag{6.18}$$

and so the trajectory is noisy in practice.

As we shall see in chapter 7, the gradient magnitudes of neural network parameters can depend on their depth in the network. Adam helps compensate for this tendency and balances out changes across the different layers. In practice, Adam also has the advantage that it is less sensitive to the initial learning rate, because it avoids situations like those in figures 6.9a–b, and so it doesn't need complex learning rate schedules.

6.5 Training algorithm hyperparameters

The choices of learning algorithm, batch size, learning rate schedule, and momentum coefficients are all considered *hyperparameters* of the training algorithm; these directly affect the quality of the final model but are distinct from the model parameters. Choosing these can be more art than science, and it's common to train many models with different hyperparameters and choose the best one. This is known as *hyperparameter tuning*. We return to this issue in chapter 8.

6.6 Summary

This chapter discussed model training. This problem was framed as finding the parameters $\boldsymbol{\phi}$ that corresponded to the minimum of a loss function $L[\boldsymbol{\phi}]$. The gradient descent method measures the gradient of the loss function for the current parameters (i.e., how the loss changes when we make a small change to the parameters). Then it moves the parameters in the direction that decreases the loss fastest. This is repeated until convergence.

Unfortunately, for nonlinear functions, the loss function may have both local minima (where gradient descent gets trapped) and saddle points (where gradient descent may appear to have converged). We introduced stochastic gradient descent, which helps mitigate these problems. At each iteration, we use a different random subset of the data (a batch) to compute the gradient. This adds noise to the process and helps prevent the algorithm from getting trapped in a suboptimal region of parameter space. Each iteration is also computationally cheaper. We saw that adding a momentum term makes convergence more efficient. Finally, we introduced the Adam algorithm.

The ideas in the chapter are applicable to optimizing *any* model. The next chapter tackles two aspects of training that are specific to neural networks. First, we address how to compute the gradients of the loss with respect to the parameters of a neural network. This is accomplished using the famous backpropagation algorithm. Second, we discuss how to initialize the network parameters before optimization begins. Without careful initialization, the gradients used by the optimization can become extremely large or extremely small, and this can hinder the training process.

Notes

Optimization algorithms: Optimization algorithms are used extensively throughout engineering although it is more typical to use the term *objective function* rather than loss function or cost function. Gradient descent was invented by Cauchy (1847), and stochastic gradient descent dates back to at least Robbins & Monro (1951). A modern compromise between the two is stochastic variance-reduced descent (Johnson & Zhang 2013), in which the full gradient is computed periodically, with stochastic updates interspersed. Reviews of optimization algorithms for neural networks can be found in Ruder (2016), Bottou et al. (2018), and Sun (2020).

Bottou (2012) discusses best practice for SGD, including shuffling without replacement.

Convexity, minima, and saddle points: A function is convex if no chord (line segment between two points on the surface) intersects the function. This can be tested algebraically by considering the *Hessian matrix* (the matrix of second derivatives):

$$\mathbf{H}[\boldsymbol{\phi}] = \begin{bmatrix} \frac{\partial^2 L}{\partial \phi_0^2} & \frac{\partial^2 L}{\partial \phi_0 \partial \phi_1} & \cdots & \frac{\partial^2 L}{\partial \phi_0 \partial \phi_N} \\ \frac{\partial^2 L}{\partial \phi_1 \partial \phi_0} & \frac{\partial^2 L}{\partial \phi_1^2} & \cdots & \frac{\partial^2 L}{\partial \phi_1 \partial \phi_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 L}{\partial \phi_N \partial \phi_0} & \frac{\partial^2 L}{\partial \phi_N \partial \phi_1} & \cdots & \frac{\partial^2 L}{\partial \phi_N^2} \end{bmatrix}. \tag{6.19}$$

Appendix C.1.4 Eigenvalues

If the Hessian matrix is positive definite (has positive eigenvalues) for all possible parameter values, then the function is convex; the loss function will look like a smooth bowl (as in figure 6.1c), and so training will be relatively easy. There will be a single global minimum and no local minima or saddle points.

For any loss function, the eigenvalues of the Hessian matrix at places where the gradient is zero allow us to classify this position as (i) a minimum (the eigenvalues are all positive), (ii) a maximum (the eigenvalues are all negative), or (iii) a saddle point (positive eigenvalues are associated with directions in which we are at a minimum and negative ones with directions where we are at a maximum).

Line search: Gradient descent with a fixed step size is inefficient because the distance moved depends entirely on the magnitude of the gradient. It moves a long distance when the function is changing fast (where perhaps it should be more cautious), but a short distance when the function is changing slowly (where perhaps it should explore further). For this reason, gradient descent methods are usually combined with a line search procedure in which we sample the function along the desired direction to try to find the optimal step size. One such approach is bracketing (figure 6.10). Another problem with gradient descent is that it tends to lead to inefficient oscillatory behavior when descending valleys (e.g., path 1 in figure 6.5a).

Beyond gradient descent: Numerous algorithms have been developed that remedy the problems of gradient descent. Most notable is the Newton method, which takes the curvature of the surface into account using the inverse of the Hessian matrix; if the gradient of the function is changing quickly, then it applies a more cautious update. This method eliminates the need for line search and does not suffer from oscillatory behavior. However, it has its own problems; in its simplest form, it moves toward the nearest extremum, but this may be a maximum if we are closer to the top of a hill than we are to the bottom of a valley. Moreover, computing the inverse Hessian is intractable when the number of parameters is large, as in neural networks.

Problem 6.9

Exhaustive search: All the algorithms discussed in this chapter are iterative. A completely different approach is to quantize the network parameters and to exhaustively search the resulting discretized parameter space using SAT solvers (Mezard & Mora 2008). This approach has the potential to find the global minimum and provide a guarantee that there is no lower loss elsewhere but is only practical for very small models.

Momentum: The idea of using momentum to speed up optimization dates to Polyak (1964). Goh (2017) presents an in-depth discussion of the properties of momentum. The Nesterov accelerated gradient method was introduced in Nesterov (1983). The Nesterov momentum approach was first applied in the context of batch gradient descent by Sutskever et al. (2013).

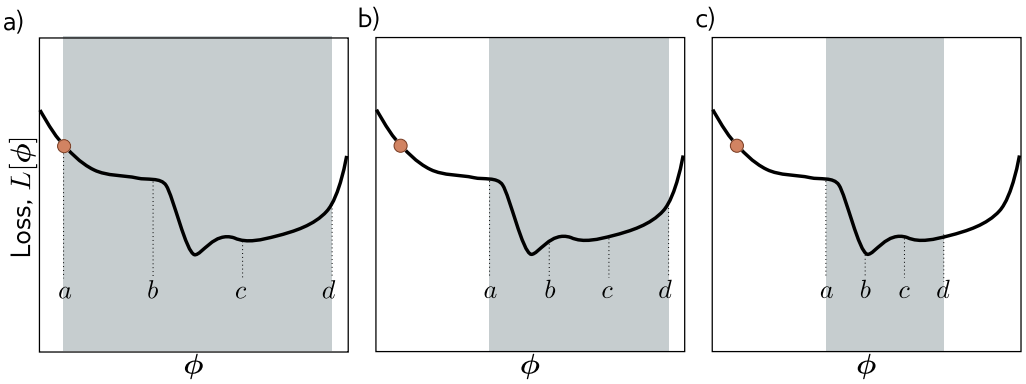

Figure 6.10 Line search using the bracketing approach. a) The current solution is at position a (orange point), and we wish to search the region $[a, d]$ (gray shaded area). We define two points b, c interior to the search region and evaluate the loss function at these points. Here $L[b] > L[c]$, and so we eliminate the range $[a, b]$. b) We now repeat this procedure in the refined search region and find that $L[b] < L[c]$, and so we eliminate the range $[c, d]$. c) We repeat this process until this minimum is closely bracketed.

Adaptive training algorithms: AdaGrad (Duchi et al. 2011) is an optimization algorithm that addresses the possibility that some parameters may have to move further than others by assigning a different learning rate to each parameter. AdaGrad uses the cumulative squared gradient for each parameter to attenuate its learning rate. This has the disadvantages that the learning rates decrease over time and that learning can halt before the minimum is found. Both RMSProp (Hinton et al. 2012a) and AdaDelta (Zeiler 2012) modified this algorithm to help prevent these problems by updating the squared gradient term recursively.

By far the most widely used adaptive training algorithm is adaptive moment optimization or Adam (Kingma & Ba 2014). This combines the ideas of momentum (in which the gradient vector is averaged over time) and AdaGrad, AdaDelta, and RMSProp (in which a smoothed squared gradient term is used to modify the learning rate for each parameter). The original paper on the Adam algorithm provided a convergence proof for convex loss functions, but a counterexample was identified by Reddi et al. (2018), who developed a modification of Adam called AMSGrad, which does converge. Of course, in deep learning the loss functions are nonconvex, and Zaheer et al. (2018) subsequently developed an adaptive algorithm called YOGI and proved that it converges in this scenario. Regardless of these theoretical objections, the original Adam algorithm works well in practice and is widely used, not least because it works well over a broad range of hyperparameters and makes rapid initial progress.

One potential problem with adaptive training algorithms is that the learning rates are based on accumulated statistics of the observed gradients. At the start of training, when there are few samples, these statistics may be very noisy. This can be remedied by *learning rate warm-up* (Goyal et al. 2018), in which the learning rates are gradually increased over the first few thousand iterations. An alternative solution is rectified Adam (Liu et al. 2021a), which gradually changes the momentum term over time in a way that helps avoid high variance. Dozat (2016) incorporated Nesterov momentum into the Adam algorithm.

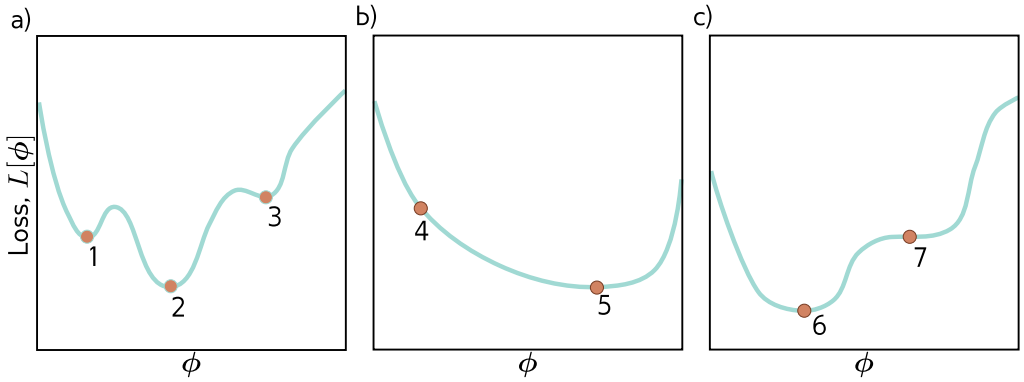

Figure 6.11 Three one dimensional loss functions for problem 6.4.

SGD vs. Adam: There has been a lively discussion about the relative merits of SGD and Adam. Wilson et al. (2018) provided evidence that SGD with momentum can find lower minima than Adam that generalize better over a variety of deep learning tasks. However, this is strange, since SGD is a special case of Adam (when $\epsilon = \gamma = 0$) once the modification term (equation 6.16) becomes one, which happens quickly. It is hence more likely that SGD outperforms Adam *when we use Adam's default hyperparameters*. Loshchilov & Hutter (2017) proposed a variation of Adam called AdamW, which substantially improves the performance of Adam in the presence of L2 regularization (see section 9.1). Choi et al. (2019) provide evidence that if we search for the best Adam hyperparameters, then it performs just as well as SGD and converges faster. Keskar & Socher (2017) proposed a method called SWATS that starts using Adam (to make rapid initial progress) and then switches to SGD (to get better final generalization performance).

Problems

Problem 6.1 Show that the derivatives of the least squares loss function in equation 6.5 are given by the expressions in equation 6.7.

Problem 6.2 A surface is convex if the eigenvalues of the Hessian $\mathbf{H}[\boldsymbol{\phi}]$ are positive everywhere. In this case, the surface has a unique minimum, and optimization is easy. Find an algebraic expression for the Hessian matrix,

$$\mathbf{H}[\boldsymbol{\phi}] = \begin{bmatrix} \frac{\partial^2 L}{\partial \phi_0^2} & \frac{\partial^2 L}{\partial \phi_0 \partial \phi_1} \\ \frac{\partial^2 L}{\partial \phi_1 \partial \phi_0} & \frac{\partial^2 L}{\partial \phi_1^2} \end{bmatrix}, \tag{6.20}$$

Appendix C.1.4 Eigenvalues
Appendix C.1.1 Trace
Appendix C.1.2 Determinant

for the linear regression model (equation 6.5). Prove that this function is convex by showing that the eigenvalues are always positive. This can be done by showing that both the trace and the determinant of the matrix are positive.

Problem 6.3 Compute the derivatives of the least squares loss $L[\boldsymbol{\phi}]$ with respect to the parameters, ϕ_0 and ϕ_1 for the Gabor model (equation 6.8).

Problem 6.4 Which of the functions in figure 6.11 is convex? Justify your answer. Characterize each of the points 1-7 as (i) a local minimum, (ii) the global minimum, or (iii) neither.

Problem 6.5 The logistic regression model uses a linear function to predict which of two classes $y \in \{0, 1\}$ an input $\mathbf{x}$ belongs to. For a 1D input and a 1D output, it has two parameters, ϕ_0 and ϕ_1, and is defined by:

$$Pr(y = 1|x) = \text{sig}[\phi_0 + \phi_1 x], \tag{6.21}$$

where sig[•] is the logistic sigmoid function:

$$\text{sig}[z] = \frac{1}{1 + \exp[-z]}. \tag{6.22}$$

(i) Plot y against x for this model for different values of ϕ_0 and ϕ_1 and explain the qualitative meaning of each parameter. (ii) What is a suitable loss function for this model? (iii) Compute the derivatives of this loss function with respect to the parameters. (iv) Generate ten data points from a normal distribution with mean minus one and standard deviation one and assign the label $y = 0$ to them. Generate another ten data points from a normal distribution with mean one and standard deviation one and assign the label $y = 1$ to these. Plot the loss as a heatmap in terms of the two parameters ϕ_0 and ϕ_1. (v) Is this loss function convex? How could you prove this?

Problem 6.6 Compute the derivatives of the least squares loss with respect to the ten parameters of the simple neural network model introduced in equation 3.1:

$$y = \phi_0 + \phi_1 \text{a}[\theta_{10} + \theta_{11} x] + \phi_2 \text{a}[\theta_{20} + \theta_{21} x] + \phi_3 \text{a}[\theta_{30} + \theta_{31} x]. \tag{6.23}$$

Think carefully about what the derivative of the ReLU function $a[\bullet]$ will be.

Problem 6.7 The gradient descent trajectory for path 1 in figure 6.5a oscillates back and forth inefficiently as it moves down the valley toward the minimum. It's also notable that it turns at right angles to the previous direction at each step. Provide a qualitative explanation for these phenomena. Propose a solution that might help prevent this behavior.

Problem 6.8 Show that the momentum term $\mathbf{m}_t$ (equation 6.11) is an infinite weighted sum of the gradients at the previous iterations and derive an expression for the coefficients (weights) of that sum.

Problem 6.9 What dimensions will the Hessian have if the model has one million parameters?

Chapter 7

Gradients and initialization

Chapter 6 introduced iterative optimization algorithms. These are general purpose methods for finding the minimum of a function. In the context of neural networks, they find parameters that minimize the loss so that the model accurately predicts the training outputs from the inputs. The basic approach is to choose initial parameters randomly, and then make a series of small changes that decrease the loss on average. Each change is based on the gradient of the loss with respect to the parameters at the current position.

This chapter discusses two issues that are specific to neural networks. First, we consider how to calculate the gradients efficiently. This is a serious challenge since the largest models at the time of writing have $\sim 10^{12}$ parameters, and the gradient needs to be computed for every parameter and at every iteration of the training algorithm. Second, we consider how to initialize the parameters. If this is not done carefully, the initial losses and their gradients can be extremely large or extremely small. In either case, this impedes the training process.

7.1 Problem definitions

Consider a network $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ with multivariate input $\mathbf{x}$, parameters $\boldsymbol{\phi}$, and three hidden layers $\mathbf{h}_1, \mathbf{h}_2$, and $\mathbf{h}_3$:

$$\begin{aligned} \mathbf{h}_1 &= \mathbf{a}[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}] \\ \mathbf{h}_2 &= \mathbf{a}[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1 \mathbf{h}_1] \\ \mathbf{h}_3 &= \mathbf{a}[\boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2 \mathbf{h}_2] \\ \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}] &= \boldsymbol{\beta}_3 + \boldsymbol{\Omega}_3 \mathbf{h}_3, \end{aligned} \tag{7.1}$$

where the function $\mathbf{a}[\bullet]$ applies the activation function separately to every element of the input. The model parameters $\boldsymbol{\phi} = \{\boldsymbol{\beta}_0, \boldsymbol{\Omega}_0, \boldsymbol{\beta}_1, \boldsymbol{\Omega}_1, \boldsymbol{\beta}_2, \boldsymbol{\Omega}_2, \beta_3, \boldsymbol{\Omega}_3\}$ consist of the bias vectors $\boldsymbol{\beta}_k$ and weight matrices $\boldsymbol{\Omega}_k$ for each of the K layers (figure 7.1).

We also have individual loss terms $l_i = \mathrm{l}[\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}], y_i]$, which return the negative log-likelihood of the ground truth label y_i given the model prediction $\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ for training input $\mathbf{x}_i$. For example, it might be the multi-class classification loss (equation 5.27). The total loss is the sum of these terms over the training data:

$$L[\boldsymbol{\phi}] = \sum_{i=1}^{I} l_i = \sum_{i=1}^{I} \mathrm{l}[\mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}], y_i]. \tag{7.2}$$

The most commonly used optimization algorithm for training neural networks is stochastic gradient descent (SGD), which updates the parameters as:

$$\boldsymbol{\phi}_{t+1} \longleftarrow \boldsymbol{\phi}_t - \alpha \sum_{i \in \mathcal{B}_t} \frac{\partial l_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}, \tag{7.3}$$

where α is the learning rate and $\mathcal{B}_t$ contains the batch indices at iteration t. To compute this update, we need to calculate terms of the form:

$$\frac{\partial l_i}{\partial \boldsymbol{\beta}_k} \quad \text{and} \quad \frac{\partial l_i}{\partial \boldsymbol{\Omega}_k}, \tag{7.4}$$

for the parameters $\{\boldsymbol{\beta}_k, \boldsymbol{\Omega}_k\}$ at every layer $k \in \{0, 1, \ldots, K\}$ and for each index i in the batch. The first part of this chapter describes the *backpropagation algorithm*, the purpose of which is to compute these derivatives efficiently.

Problem 7.1

In the second part of the chapter, we consider how to initialize the network parameters before we commence training. We describe methods to choose the initial weights $\boldsymbol{\Omega}_k$ and biases $\boldsymbol{\beta}_k$ so that training is stable.

7.2 Computing derivatives

The derivatives of the loss tell us how the loss changes when we make a small change to the parameters. Optimization algorithms exploit this information to manipulate the parameters so that the loss becomes smaller. The *backpropagation algorithm* computes these derivatives. The mathematical details are somewhat involved, and so before describing them, we first make two observations that provide some intuition.

Observation 1: Each weight (element of $\boldsymbol{\Omega}_k$) multiplies the activation at a source hidden unit and adds the result to a destination hidden unit in the next layer. It follows that the effect of any small change to the weight is amplified or attenuated by the activation at the source hidden unit. Hence, we run the network for each data example in the batch and store the activations at all of the hidden units. This is known as the *forward pass* (figure 7.1). The stored activations will subsequently be used to compute the gradients.

Observation 2: A small change in a bias or weight causes a ripple effect of changes through the subsequent network. The change modifies the value of its destination hidden

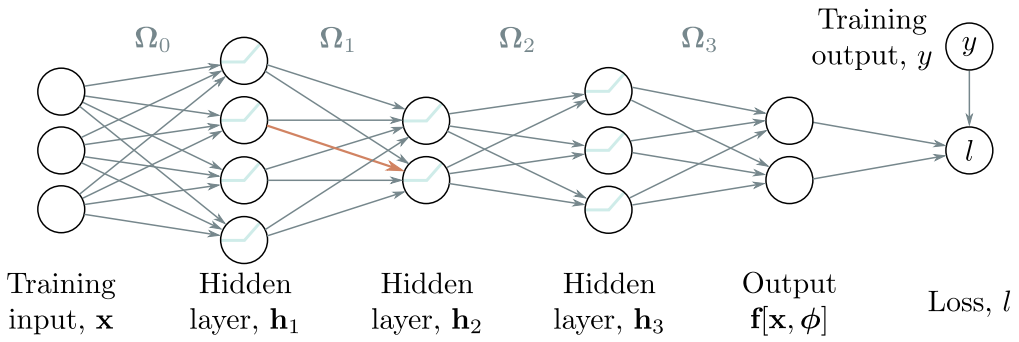

Figure 7.1 Backpropagation forward pass. The goal is to compute the derivatives of the loss l with respect to each of the weights (arrows) and biases (not shown). In other words, we want to know how a small change to each parameter will affect the loss. Each weight multiplies the hidden unit at its source and contributes the result to the hidden unit at its destination. Consequently, the effects of any small change to the weight will be scaled by the activation of the source hidden unit. For example, the orange weight is applied to the second hidden unit at layer 1; if the activation of this unit doubles, then the effect of a small change to the orange weight will double too. It follows that to compute the derivatives of the weights, we need to know the activations at the hidden layers. Computing and storing these is referred to as the *forward pass*, since it involves running the network equations sequentially for the data example.

unit. This in turn changes the values of the hidden units in the subsequent layer, which will change the hidden units in the layer after that and so on, until a change is made to the model output and finally the loss.

Hence, to know how changing a parameter modifies the loss, we also need to know how changes to every subsequent hidden layer will in turn modify their successor. These same quantities are needed when we consider other parameters in the same layer or parameters in earlier layers. It follows that we can calculate them once and re-use them. For example, consider computing the effect of a small change in weights that feed into hidden layers $\mathbf{h}_3$, $\mathbf{h}_2$, and $\mathbf{h}_1$ respectively:

- To calculate how a small change in a weight or bias feeding into hidden layer $\mathbf{h}_3$ modifies the loss, we need to know (i) how a change in layer $\mathbf{h}_3$ changes the model output $\mathbf{f}$, and (ii) how a change in model output changes the loss l (figure 7.2a).
- To calculate how a small change in a weight or bias feeding into hidden layer $\mathbf{h}_2$ modifies the loss, we need to know (i) how a change in layer $\mathbf{h}_2$ affects $\mathbf{h}_3$, (ii) how $\mathbf{h}_3$ changes the model output, and (iii) how this output changes the loss (figure 7.2b).
- To calculate how a small change in a weight or bias feeding into hidden layer $\mathbf{h}_1$ modifies the loss, we need to know (i) how a change in layer $\mathbf{h}_1$ affects layer $\mathbf{h}_2$,

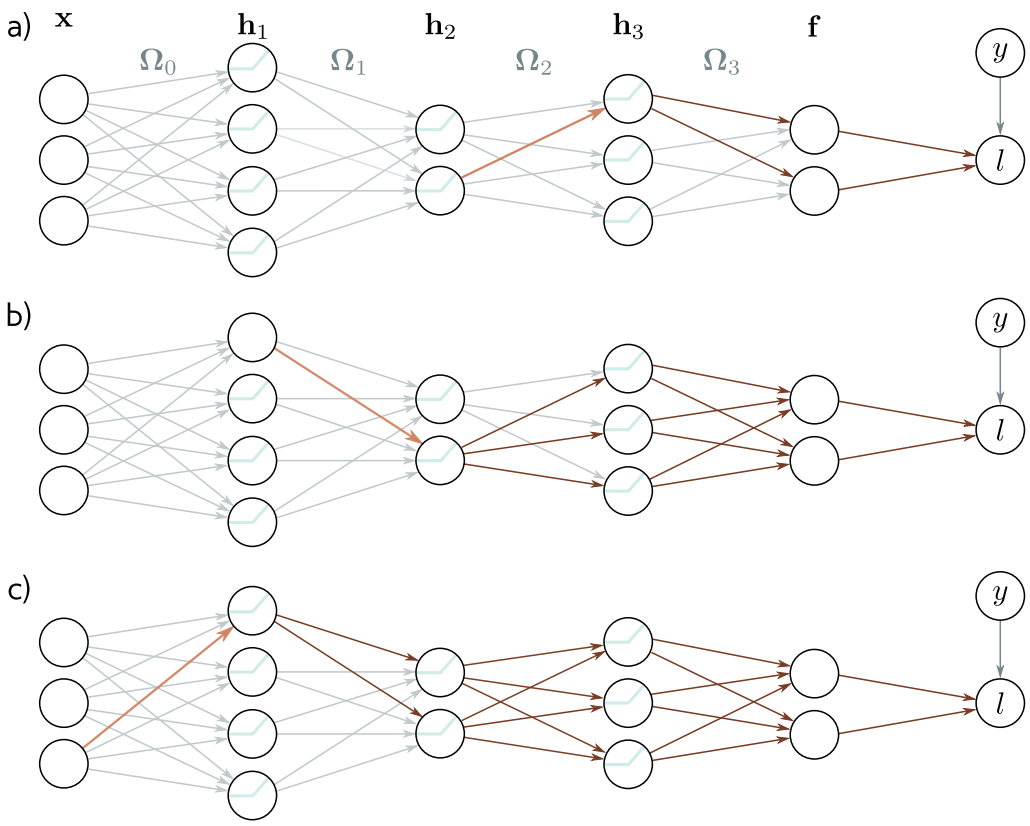

Figure 7.2 Backpropagation backward pass. a) To compute how a change to a weight feeding into layer $\mathbf{h}_3$ (orange arrow) changes the loss, we need to know how the hidden unit in $\mathbf{h}_3$ changes the model output $\mathbf{f}$ and how $\mathbf{f}$ changes the loss (brown arrows). b) To compute how a small change to a weight feeding into $\mathbf{h}_2$ (orange arrow) changes the loss, we need to know (i) how the hidden unit in $\mathbf{h}_2$ changes $\mathbf{h}_3$, (ii) how $\mathbf{h}_3$ changes $\mathbf{f}$, and (iii) how $\mathbf{f}$ changes the loss (brown arrows). c) Similarly, to compute how a small change to a weight feeding into $\mathbf{h}_1$ (orange arrow) changes the loss, we need to know how $\mathbf{h}_1$ changes $\mathbf{h}_2$ and how these changes propagate through to the loss (brown arrows). The backward pass computes derivatives at the end of the network first and works backward to exploit the inherent redundancy of these computations.

(ii) how a change in layer $\mathbf{h}_2$ affects layer $\mathbf{h}_3$, (iii) how layer $\mathbf{h}_3$ changes the model output, and (iv) how the model output changes the loss (figure 7.2c).

As we move backward through the network, we see that most of the terms that we need have already been computed in the previous step. and so we can re-use this computation. Proceeding backward through the network in this way to compute the derivatives is known as the *backward pass*.

7.2.1 Backpropagation

The backpropagation algorithm follows from these two intuitions and consists of (i) a forward pass, in which we compute and store the values at all of the hidden units and the network output, and (ii) a backward pass, in which we calculate the derivatives of each parameter, starting at the end of the network, and reusing the previous computation as we move towards the start.

Forward pass: We can think of the network in equation 7.1 as a series of sequential calculations:

$$\begin{aligned}
\mathbf{f}_0 &= \boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0\mathbf{x}_i \\
\mathbf{h}_1 &= \mathbf{a}[\mathbf{f}_0] \\
\mathbf{f}_1 &= \boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1\mathbf{h}_1 \\
\mathbf{h}_2 &= \mathbf{a}[\mathbf{f}_1] \\
\mathbf{f}_2 &= \boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2\mathbf{h}_2 \\
\mathbf{h}_3 &= \mathbf{a}[\mathbf{f}_2] \\
\mathbf{f}_3 &= \boldsymbol{\beta}_3 + \boldsymbol{\Omega}_3\mathbf{h}_3 \\
l_i &= \mathrm{l}[\mathbf{f}_3, y_i],
\end{aligned} \tag{7.5}$$

where $\mathbf{f}_{k-1}$ represents the pre-activations at the k^{th} hidden layer (i.e., the values before the ReLU function $\mathbf{a}[\bullet]$) and $\mathbf{h}_k$ contains the activations at the k^{th} hidden layer (i.e., after the ReLU function). In the forward pass, we simply work through these calculations and store all the intermediate quantities.

Backward pass #1: Now let's consider how the loss changes when we modify the pre-activations $\mathbf{f}_0, \mathbf{f}_1, \mathbf{f}_2$. Applying the chain rule, the expression for the derivative of the loss l_i with respect to $\mathbf{f}_2$ is:

Matrix calculus
Appendix C.2

$$\frac{\partial l_i}{\partial \mathbf{f}_2} = \frac{\partial \mathbf{h}_3}{\partial \mathbf{f}_2}\frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_3}\frac{\partial l_i}{\partial \mathbf{f}_3}. \tag{7.6}$$

The three terms on the right-hand side have sizes $D_3 \times D_3, D_3 \times D_f$, and $D_f \times 1$ respectively, where D_3 is the number of hidden units in the third layer and D_f is the dimensionality of the model output $\mathbf{f}_3$.

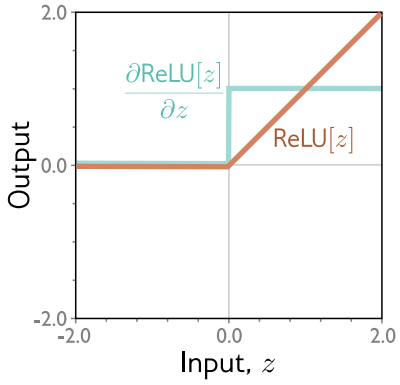

Figure 7.3 Derivative of rectified linear unit. The rectified unit (orange curve) returns zero when the input is less than zero and returns the input otherwise. Its derivative (cyan curve) returns zero when the input is less than zero (since the slope here is zero) and one when the input is greater than zero (since the slope here is one).

Similarly, we can compute how the loss changes when we change $\mathbf{f}_1$ and $\mathbf{f}_0$:

$$\frac{\partial l_i}{\partial \mathbf{f}_1} = \frac{\partial \mathbf{h}_2}{\partial \mathbf{f}_1}\frac{\partial \mathbf{f}_2}{\partial \mathbf{h}_2}\left(\frac{\partial \mathbf{h}_3}{\partial \mathbf{f}_2}\frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_3}\frac{\partial l_i}{\partial \mathbf{f}_3}\right) \tag{7.7}$$

$$\frac{\partial l_i}{\partial \mathbf{f}_0} = \frac{\partial \mathbf{h}_1}{\partial \mathbf{f}_0}\frac{\partial \mathbf{f}_1}{\partial \mathbf{h}_1}\left(\frac{\partial \mathbf{h}_2}{\partial \mathbf{f}_1}\frac{\partial \mathbf{f}_2}{\partial \mathbf{h}_2}\frac{\partial \mathbf{h}_3}{\partial \mathbf{f}_2}\frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_3}\frac{\partial l_i}{\partial \mathbf{f}_3}\right). \tag{7.8}$$

Problem 7.2

Note that in each case, the term in brackets was computed in the previous step. By working backward through the network, we can re-use the previous computations.

Problems 7.3-7.4

Moreover, the terms themselves are extremely simple. Working backward through the right-hand side of equation 7.6, we have:

- The derivative $\partial l_i/\partial \mathbf{f}_3$ of the loss l_i with respect to the network output $\mathbf{f}_3$ will depend on the loss function but usually has a simple form.

Problem 7.5

- The derivative $\partial \mathbf{f}_3/\partial \mathbf{h}_3$ of the network output with respect to hidden layer $\mathbf{h}_3$ is:

$$\frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_3} = \frac{\partial}{\partial \mathbf{h}_3}\left(\boldsymbol{\beta}_3 + \boldsymbol{\Omega}_3\mathbf{h}_3\right) = \boldsymbol{\Omega}_3^T. \tag{7.9}$$

  If you are not familiar with matrix calculus, then this result is not obvious. It is explored in problem 7.5.

Problems 7.6-7.7

- The derivative $\partial \mathbf{h}_3/\partial \mathbf{f}_2$ of the output $\mathbf{h}_3$ of the activation function with respect to its input $\mathbf{f}_2$ will depend on the activation function. For ReLU functions, this is a diagonal matrix where the diagonal terms are zero everywhere $\mathbf{f}_2$ is less than zero and one everywhere it is greater (figure 7.3). Rather than multiply by this matrix, we extract the diagonal terms as a vector $\mathbb{I}[\mathbf{f}_2 > 0]$ and pointwise multiply, which is more efficient.

The terms on the right-hand side of equations 7.7 and 7.8 have similar forms. As we progress back through the network, we alternately (i) multiply by the transpose of the weight matrices $\boldsymbol{\Omega}_k^T$ and (ii) threshold based on the inputs $\mathbf{f}_{k-1}$ to the hidden layer. These inputs were stored during the forward pass.

Backward pass #2: Now that we know how to compute the derivatives $\partial l_i/\partial \mathbf{f}_k$, we can focus on calculating the derivatives of the loss with respect to the weights and biases. To calculate the derivatives of the loss with respect to the biases $\boldsymbol{\beta}_k$ we again use the chain rule:

$$\begin{aligned}\frac{\partial l_i}{\partial \boldsymbol{\beta}_k} &= \frac{\partial \mathbf{f}_k}{\partial \boldsymbol{\beta}_k}\frac{\partial l_i}{\partial \mathbf{f}_k} \\ &= \frac{\partial}{\partial \boldsymbol{\beta}_k}\left(\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k\mathbf{h}_k\right)\frac{\partial l_i}{\partial \mathbf{f}_k} \\ &= \frac{\partial l_i}{\partial \mathbf{f}_k},\end{aligned} \tag{7.10}$$

which we already calculated in equations 7.6 and 7.7.

Similarly, the derivative for the weights vector $\boldsymbol{\Omega}_k$, is given by:

$$\begin{aligned}\frac{\partial l_i}{\partial \boldsymbol{\Omega}_k} &= \frac{\partial \mathbf{f}_k}{\partial \boldsymbol{\Omega}_k}\frac{\partial l_i}{\partial \mathbf{f}_k} \\ &= \frac{\partial}{\partial \boldsymbol{\Omega}_k}\left(\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k\mathbf{h}_k\right)\frac{\partial l_i}{\partial \mathbf{f}_k} \\ &= \frac{\partial l_i}{\partial \mathbf{f}_k}\mathbf{h}_k^T.\end{aligned} \tag{7.11}$$

Again, the progression from line two to line three is not obvious and is explored in problem 7.8. However, the result makes sense. The final line is a matrix of the same size as $\boldsymbol{\Omega}_k$. It depends linearly on $\mathbf{h}_k$, which was multiplied by $\boldsymbol{\Omega}_k$ in the original expression. This is also consistent with the initial intuition that the derivative of the weights in $\boldsymbol{\Omega}_k$ will be proportional to the values of the hidden units $\mathbf{h}_k$ that they multiply. Recall that we already computed these during the forward pass.

Problem 7.8

7.2.2 Backpropagation algorithm summary

We'll now briefly summarize the final backpropagation algorithm. Consider a deep neural network $\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ that takes input $\mathbf{x}_i$, has K hidden layers with ReLU activations, and individual loss term $l_i = \mathrm{l}[\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}], \mathbf{y}_i]$. The goal of backpropagation is to compute the derivatives $\partial l_i/\partial \boldsymbol{\beta}_k$ and $\partial l_i/\partial \boldsymbol{\Omega}_k$ with respect to the weights $\boldsymbol{\Omega}_k$ and biases $\boldsymbol{\beta}_k$.

Forward pass: We compute and store the following quantities:

$$\begin{aligned}\mathbf{f}_0 &= \boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0\mathbf{x}_i \\ \mathbf{h}_k &= \mathbf{a}[\mathbf{f}_{k-1}] \qquad\qquad k \in \{1, 2, \ldots K\} \\ \mathbf{f}_k &= \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k\mathbf{h}_k. \qquad\qquad k \in \{1, 2, \ldots K\}\end{aligned} \tag{7.12}$$

Backward pass: We start with the derivative $\partial l_i/\partial \mathbf{f}_K$ of the loss function l_i with respect to the network output $\mathbf{f}_K$ and work backward through the network:

$$\begin{aligned}
\frac{\partial l_i}{\partial \boldsymbol{\beta}_k} &= \frac{\partial l_i}{\partial \mathbf{f}_k} && k \in \{K, K-1, \ldots 1\} \\
\frac{\partial l_i}{\partial \boldsymbol{\Omega}_k} &= \frac{\partial l_i}{\partial \mathbf{f}_k}\mathbf{h}_k^T && k \in \{K, K-1, \ldots 1\} \\
\frac{\partial l_i}{\partial \mathbf{f}_{k-1}} &= \mathbb{I}[\mathbf{f}_{k-1} > 0] \odot \left(\boldsymbol{\Omega}_k^T \frac{\partial l_i}{\partial \mathbf{f}_k}\right), && k \in \{K, K-1, \ldots 1\}
\end{aligned} \tag{7.13}$$

where $\odot$ denotes pointwise multiplication and $\mathbb{I}[\mathbf{f}_{k-1} > 0]$ is a vector containing ones where $\mathbf{f}_{k-1}$ is greater than zero and zeros elsewhere. Finally, we compute the derivatives with respect to the first set of biases and weights:

$$\begin{aligned}
\frac{\partial l_i}{\partial \boldsymbol{\beta}_0} &= \frac{\partial l_i}{\partial \mathbf{f}_0} \\
\frac{\partial l_i}{\partial \boldsymbol{\Omega}_0} &= \frac{\partial l_i}{\partial \mathbf{f}_0}\mathbf{x}_i^T.
\end{aligned} \tag{7.14}$$

Problems 7.9-7.10

We calculate these derivatives for every training example in the batch and sum them together to retrieve the gradient for the stochastic gradient descent update.

7.2.3 Algorithmic differentiation

Although it's important to understand the backpropagation algorithm, it's unlikely that you will need to code it in practice. Modern deep learning frameworks such as PyTorch and Tensorflow calculate the derivatives automatically given the model specification. This is known as *algorithmic differentiation*.

Each functional component (linear transform, ReLU activation, loss function) in the framework knows how to compute its own derivative. For example, the PyTorch ReLU function $\mathbf{z}_{out} = \mathbf{relu}[\mathbf{z}_{in}]$ knows how to compute the derivative of its output $\mathbf{z}_{out}$ with respect to its input $\mathbf{z}_{in}$. Similarly, a linear function $\mathbf{z}_{out} = \boldsymbol{\beta} + \boldsymbol{\Omega}\mathbf{z}_{in}$ knows both how to compute the derivatives of the output $\mathbf{z}_{out}$ with respect to the input $\mathbf{z}_{in}$ and with respect to the parameters $\boldsymbol{\beta}$ and $\boldsymbol{\Omega}$. The algorithmic differentiation framework also knows the sequence of operations in the network and thus has all the information required to perform the forward and backward passes.

These frameworks exploit the massive parallelism of modern graphic processor units (GPUs). Computations such as matrix multiplication (which features in both the forward and backward pass) are naturally amenable to parallelization. Moreover, it's possible to perform the forward and backward passes for the entire batch in parallel, if the model and intermediate results in the forward pass do not exceed the available memory.

Problem 7.11

Since the training algorithm now processes the entire batch in parallel, the input becomes a multi-dimensional *tensor*. In this context, a tensor can be considered the generalization of a matrix to arbitrary dimensions. Hence, a vector is a 1D tensor, a

matrix is a 2D tensor, and a 3D tensor is a 3D grid of numbers. Until now, the training data have been 1D, and so the input for backpropagation would be a 2D tensor where the first dimension is the data index and the second is the batch index. In subsequent chapters, we will encounter more complex structured input data. For example, in models where the input is an RGB image, the original data examples are 3D (height $\times$ width $\times$ channel). Here, the input to the learning framework would be a 4D tensor, where the last dimension indexes the batch element.

7.2.4 Extension to arbitrary computational graphs

In the previous section, we described backpropagation in a deep neural network; this model is naturally sequential and we calculate the intermediate quantities $\mathbf{f}_0, \mathbf{h}_1, \mathbf{f}_1, \mathbf{h}_2 \ldots \mathbf{f}_k$ in turn. However, models need not be restricted to sequential computation. Later in this book, we will meet models with branching structures. For example, we might take the values in a hidden layer and process them through two different sub-networks before recombining.

Problems 7.12-7.13

Fortunately, the ideas of backpropagation still hold if the computational graph is acyclic. Modern algorithmic differentiation frameworks such as PyTorch and Tensorflow can handle arbitrary acyclic computational graphs.

7.3 Parameter initialization

We have discussed both stochastic gradient descent, and how to compute the derivatives that it requires. We now address how to initialize the parameters before we start training. To see why this is important, consider that during the forward pass, each hidden layer $\mathbf{h}_k$ is computed as:

$$\mathbf{h}_{k+1} = \mathbf{a}[\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{h}_k],$$

where $\mathbf{a}[\bullet]$ applies the ReLU functions and $\boldsymbol{\Omega}_k$ and $\boldsymbol{\beta}_k$ are the weights and biases respectively. Imagine that we initialize all the biases to zero, and the elements of $\boldsymbol{\Omega}_k$ according to a normal distribution with mean zero and variance σ^2. Let's now consider two scenarios:

- If the variance σ^2 is very small (e.g., 10^{-5}), then each element of $\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{h}_k$ will be a weighted sum of $\mathbf{h}_k$ where the weights are very small; the result is likely to have a smaller magnitude than the input. After passing through the ReLU function, values less than zero will be clipped and the range of outputs will halve. Consequently, the magnitudes of the activations at the hidden layers will tend to get smaller and smaller as we progress through the network.
- If the variance σ^2 is very large (e.g., 10^5), then each element of $\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{h}_k$ will be a weighted sum of $\mathbf{h}_k$ where the weights are very large; the result is likely to

have a much larger magnitude than the input. After passing through the ReLU function, any values less than zero will be clipped and so the range of outputs will be halved; however, even after this, their magnitude might still be larger on average. Consequently, the magnitudes of the activations at the hidden layers will tend to get larger and larger as we progress through the network.

In these two situations, the values at the hidden layers can become so small or so large that they cannot be represented with finite precision floating point math.

Even if the forward pass is tractable, the same logic applies to the backward pass. Each gradient update (equation 7.13) consists of multiplying by $\boldsymbol{\Omega}^T$, and if the values of $\boldsymbol{\Omega}$ are not initialized sensibly, then the gradient magnitudes may decrease or increase uncontrollably during the backward pass. These cases are known as the *vanishing gradient problem* and the *exploding gradient problem*, respectively. In the former case, updates to the model become vanishingly small. In the latter case, the updates become unstable.

Problem 7.14

7.3.1 Initialization for forward pass

We now present a mathematical version of the same argument. Consider the computation between adjacent hidden layers $\mathbf{h}$ and $\mathbf{h}'$ of a neural network with dimensions D_h and $D_{h'}$, respectively:

$$\begin{aligned}\mathbf{f} &= \boldsymbol{\beta} + \boldsymbol{\Omega}\mathbf{h} \\ \mathbf{h}' &= \mathbf{a}[\mathbf{f}],\end{aligned}$$

where $\mathbf{f}$ represents the pre-activations, $\boldsymbol{\Omega}$, and $\boldsymbol{\beta}$ represent the weights and biases, and $\mathbf{a}[\bullet]$ is the activation function.

Assume the hidden units h_j in the input layer $\mathbf{h}$ have variance σ_h^2. Consider initializing the biases β_i as zero, and the weights Ω_{ij} as normally distributed with mean zero and variance σ_Ω^2. Now we'll derive expressions for the mean and variance of the intermediate values $\mathbf{f}$ and the subsequent hidden layer $\mathbf{h}'$.

Appendix B.2 Expectation operator

The mean $\mathbb{E}[f_i]$ of the intermediate values f_i is:

$$\begin{aligned}\mathbb{E}[f_i] &= \mathbb{E}\left[\beta_i + \sum_{j=1}^{D_h}\Omega_{ij}h_j\right] \\ &= \mathbb{E}\left[\beta_i\right] + \sum_{j=1}^{D_h}\mathbb{E}\left[\Omega_{ij}h_j\right] \\ &= \mathbb{E}\left[\beta_i\right] + \sum_{j=1}^{D_h}\mathbb{E}\left[\Omega_{ij}\right]\mathbb{E}\left[h_j\right] \\ &= 0 + \sum_{j=1}^{D_h} 0 \cdot \mathbb{E}\left[h_j\right] = 0, \qquad (7.15)\end{aligned}$$

where D_h is the dimensionality of the input layer $\mathbf{h}$ and we have assumed that the distributions over the hidden units h_j and the network weights Ω_{ij} are independent between the second and third lines.

Using this result, we see that the variance σ_f^2 of the intermediate values f_i is:

$$\begin{aligned}\sigma_f^2 &= \mathbb{E}[f_i^2] - \mathbb{E}[f_i]^2 \\ &= \mathbb{E}\left[\left(\sum_{j=1}^{D_h}\Omega_{ij}h_j\right)^2\right] - 0 \\ &= \sum_{j=1}^{D_h}\mathbb{E}\left[\Omega_{ij}^2\right]\mathbb{E}\left[h_j^2\right] \\ &= \sum_{j=1}^{D_h}(\sigma_\Omega^2\sigma_h^2) = D_h\sigma_\Omega^2\sigma_h^2, \end{aligned} \tag{7.16}$$

where we have used the standard identity $\sigma^2 = \mathbb{E}[(z-\mathbb{E}[z])^2] = \mathbb{E}[z^2] - \mathbb{E}[z]^2$. We have assumed once more that the distributions of the weights Ω_{ij} and the hidden units h_j are independent between lines two and three.

Appendix B.2.2 Variance identity

The initial distribution of the weights Ω_{ij} was symmetric about zero, and so the distribution of f_j will also be symmetric about zero. It follows that half of the weights will be clipped by the ReLU function and so the variance of $\mathbf{h}'$ will be half that of $\mathbf{f}$:

Problem 7.15

$$\sigma_{h'}^2 = \frac{1}{2}\sigma_f^2 = \frac{1}{2}D_h\sigma_\Omega^2\sigma_h^2. \tag{7.17}$$

This in turn implies that if we want the variance $\sigma_{h'}^2$ of the subsequent layer $\mathbf{h}'$ to be the same as the variance of the original layer $\mathbf{h}$ during the forward pass, we should set:

$$\sigma_\Omega^2 = \frac{2}{D_h}, \tag{7.18}$$

where D_h is the dimension of the original layer that the weights were applied to. This is known as *He initialization*.

7.3.2 Initialization for backward pass

A similar argument establishes how the variance of the gradients $\partial l/\partial f_k$ changes during the backward pass. During the backward pass, we multiply by the transpose $\mathbf{\Omega}^T$ of the weight matrix (equation 7.13) and so the equivalent expression becomes:

$$\sigma_\Omega^2 = \frac{2}{D_{h'}}, \tag{7.19}$$

where $D_{h'}$ is the dimension of the layer that the weights feed into.

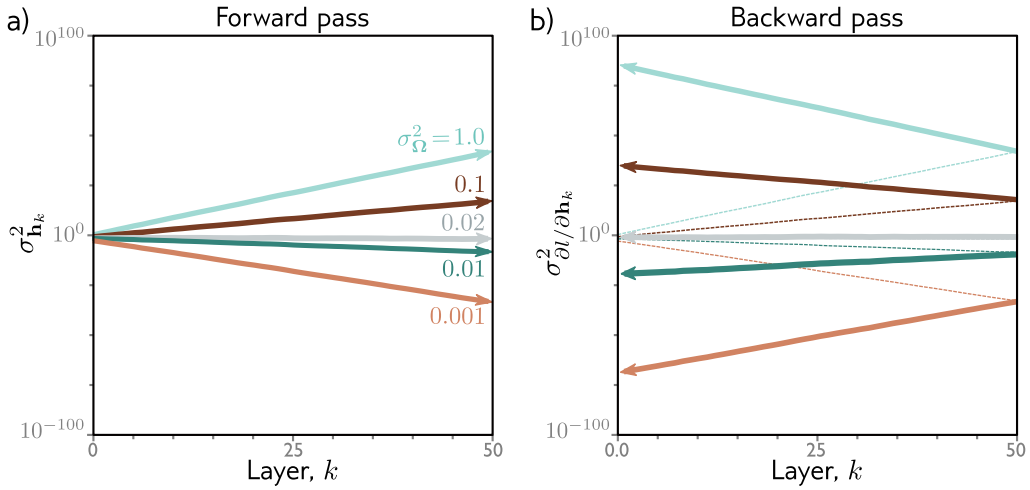

Figure 7.4 Weight initialization. Consider a deep network with 50 hidden layers and $D_h = 100$ hidden units per layer. The network has a 100 dimensional input $\mathbf{x}$ initialized with values from a standard normal distribution, a single output fixed at $y = 0$, and a least squares loss function. The bias vectors $\boldsymbol{\beta}_k$ are initialized to zero and the weight matrices $\boldsymbol{\Omega}_k$ are initialized with a normal distribution with mean zero and five different variances $\sigma^2_\Omega \in \{0.001, 0.01, 0.02, 0.1, 1.0\}$. a) Variance of hidden unit activations computed in forward pass as a function of the network layer. For He initialization ($\sigma^2_\Omega = 2/D_h = 0.02$), the variance is stable. However, for larger values it increases rapidly, and for smaller values, it decreases rapidly. b) The variance of the gradients in the backward pass (solid lines) continues this trend; if we initialize with a value larger than 0.02, the magnitude of the gradients increases rapidly as we pass back through the network. If we initialize with a value smaller, then the magnitude decreases. These are known as the *exploding gradient* and *vanishing gradient* problems, respectively.

7.3.3 Initialization for both forward and backward pass

If the weight matrix $\boldsymbol{\Omega}$ is not square (i.e., there are different numbers of hidden units in the two adjacent layers and so D_h and $D_{h'}$ differ), then it is not possible to choose the variance to satisfy both equations 7.18 and 7.19 simultaneously. One possible compromise is to use the mean $(D_h + D_{h'})/2$ as a proxy for the number of terms which gives:

$$\sigma^2_\Omega = \frac{1}{D_h + D_{h'}}. \tag{7.20}$$

Figure 7.4 shows empirically that both the variance of the hidden units in the forward pass and the variance of the gradients in the backward pass remain stable when the parameters are initialized appropriately.

```
import torch, torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader
from torch.optim.lr_scheduler import StepLR

# define input size, hidden layer size, output size
D_i, D_k, D_o = 10, 40, 5
# create model with two hidden layers
model = nn.Sequential(
   nn.Linear(D_i, D_k),
   nn.ReLU(),
   nn.Linear(D_k, D_k),
   nn.ReLU(),
   nn.Linear(D_k, D_o))

# He initialization of weights
def weights_init(layer_in):
   if isinstance(layer_in, nn.Linear):
   nn.init.kaiming_uniform(layer_in.weight)
   layer_in.bias.data.fill_(0.0)
model.apply(weights_init)

# choose least squares loss function
criterion = nn.MSELoss()
# construct SGD optimizer and initialize learning rate and momentum
optimizer = torch.optim.SGD(model.parameters(), lr = 0.01, momentum=0.9)
# object that decreases learning rate by half every 10 epochs
scheduler = StepLR(optimizer, step_size=10, gamma=0.5)

# create 100 dummy data points and store in data loader class
x = torch.randn(100, D_i)
y = torch.randn(100, D_o)
data_loader = DataLoader(TensorDataset(x,y), batch_size=10, shuffle=True)

 # loop over the dataset 10 times
for epoch in range(100):
   epoch_loss = 0.0
   # loop over batches
   for i, data in enumerate(data_loader):
   # retrieve inputs and labels for this batch
   x_batch, y_batch = data
   # zero the parameter gradients
   optimizer.zero_grad()
   # forward pass
   pred = model(x_batch)
   loss = criterion(pred, y_batch)
   # backward pass
   loss.backward()
   # SGD update
   optimizer.step()
   # update statistics
   epoch_loss += loss.item()
   # print error
   print('Epoch %5d, loss: %.3f' %(epoch, epoch_loss))
   # tell scheduler to consider updating learning rate
   scheduler.step()
```

Figure 7.5 Sample code for training two-layer network on random data.

7.4 Example training code

Problems 7.16-7.17

The primary focus of this book is scientific; this is not a guide for how to implement deep learning models. Nonetheless, in figure 7.5 we present working PyTorch code that implements the ideas explored in this book so far. The code defines a neural network and initializes the weights. It creates a random set of input and output data and defines a least squares loss function. The model is trained from the data using stochastic gradient descent with momentum in batches of size 10 over 100 epochs. The learning rate starts at 0.01 and halves every 10 epochs.

The takeaway is that although the underlying ideas in deep learning are quite complex, implementation is relatively simple. For example, all of the details of the backpropagation are hidden in the single line of code: `loss.backward()`.

7.5 Summary

The previous chapter introduced stochastic gradient descent, which is an iterative optimization algorithm that aims to find the minimum of a function. In the context of neural networks, this algorithm is used to find the parameters that minimize the loss function. Stochastic gradient descent relies on the gradient of the loss function with respect to the parameters and these parameters must be initialized before optimization. This chapter has addressed these two problems for deep neural networks.

The gradients must be evaluated for a very large number of parameters, for each member of the batch, and at each SGD iteration. It is hence imperative that the gradient computation is efficient and to this end, the backpropagation algorithm was introduced. Careful parameter initialization is also critical. The magnitudes of the hidden unit activations can either decrease or increase exponentially in the forward pass. The same is true of the gradient magnitudes in the backward pass where these behaviors are known as the disappearing gradient and exploding gradient problems. Both impede training but can be avoided with appropriate initialization.

We've now reached a point where we have defined the model and the loss function and can train a model for a given task. The following chapter discusses how to measure the performance of that model.

Notes

Backpropagation: Efficient re-use of partial computations while calculating gradients in computational graphs has been repeatedly discovered, including by Werbos (1974), Bryson et al. (1979), Lecun (1985), and Parker (1985). However, the most celebrated description of this idea was by Rumelhart et al. (1985) and Rumelhart et al. (1986) who also coined the term "backpropagation". This latter work kick-started a new phase of neural network research in the eighties and nineties; for the first time, it was practical to train networks with hidden layers. However, progress stalled due (in retrospect) to a lack of training data, limited computational power,

and the use of sigmoid activations. Areas such as natural language processing and computer vision did not rely on neural network models until the remarkable image classification results of Krizhevsky et al. (2012) ushered in the modern era of deep learning.

The implementation of backpropagation in modern deep learning frameworks such as PyTorch and Tensorflow is an example of reverse mode algorithmic differentiation. This is distinguished from forward mode algorithmic differentiation in which the derivatives from the chain rule are accumulated while moving forward through the computational graph (see problem 7.13). Further information about algorithmic differentiation can be found in Griewank & Walther (2008) and Baydin et al. (2018).

Initialization: He initialization was first introduced in He et al. (2015). It follows closely from *Glorot* or *Xavier* initialization (Glorot & Bengio 2010), which is very similar but does not consider the effect of the ReLU layer and so differs by a factor of two. Essentially the same method was proposed much earlier by LeCun et al. (1998b) but with a slightly different motivation; in this case, sigmoidal activation functions were used, which naturally normalize the range of outputs at each layer and hence help prevent an exponential increase in the magnitudes of the hidden units. However, if the inputs are too large, then they fall into the flat regions of the sigmoid function and result in very small gradients. Hence, it is still important to initialize the weights sensibly. Klambauer et al. (2017) introduce the scaled exponential unit and show that within a certain range of inputs, this activation function tends to make the activations in network layers automatically converge to mean zero and unit variance.

A completely different approach is to pass data through the network, and then normalize by the empirically observed variance. *Layer-sequential unit variance initialization* (Mishkin & Matas 2016) is an example of this kind of method, in which the weight matrices as initialized as orthonormal. GradInit (Zhu et al. 2021) randomizes the initial weights and temporarily fixes them while it learns non-negative scaling factors for each weight matrix. These factors are selected so that they maximize the decrease in the loss for a fixed learning rate subject to a constraint on the maximum norm of the gradient. *Activation normalization* or *ActNorm* adds a learnable scaling and offset parameter after each layer of the network at each hidden unit. They run an initial batch through the network, and then choose the offset and scale so that the mean of the activations is zero and the variance one. After this, these extra parameters are learned as part of the model. Closely related to these methods are schemes such as *BatchNorm* (Ioffe & Szegedy 2015), in which the network normalizes the variance of each batch as part of its processing at every step. BatchNorm and its variants are discussed in chapter 11. Other initialization schemes have been proposed for specific architectures, including the *ConvolutionOrthogonal* initializer (Xiao et al. 2018) for convolutional networks, *Fixup* (Zhang et al. 2019a) for residual networks and *TFixup* (Huang et al. 2020a), and *DTFixup* (Xu et al. 2021b) for transformers.

Reducing memory requirements: Training neural networks is demanding in terms of memory. Not only must we store the model parameters, but we must also store the pre-activations at the hidden units for every member of the batch during the forward pass. Two methods that decrease memory requirements are *gradient checkpointing* (Chen et al. 2016) and *micro-batching* (Huang et al. 2019). In gradient checkpointing, the activations are only stored every N layers during the forward pass. During the backward pass, the intermediate missing activations are recalculated from the nearest checkpoint. In this manner, we can drastically reduce the memory requirements at the computational cost of performing the forward pass twice (problem 7.11). In micro-batching, the batch is sub-divided into smaller parts and the gradient updates are aggregated from each sub-batch before being applied to the network. A completely different approach is to build a reversible network (e.g., Gomez et al. 2017), in which the previous layer can be computed from the current one and so there is no need to cache anything during the forward pass (see chapter 16). These methods and other approaches to reducing memory requirements are reviewed in Sohoni et al. (2019).

Distributed training: For sufficiently large models, the memory requirements or total required time may be too much for a single processor. In this case, we must use *distributed training*, in which training takes place in parallel across multiple processors. There are several approaches to parallelism. In *data parallelism*, each processor or *node* contains a full copy of the model but runs a subset of the batch (see Xing et al. 2015, Li et al. 2020b). The gradients from each node are aggregated centrally and then redistributed back to each node to ensure that the models remain consistent. This is known as *synchronous training*. The synchronization required to aggregate and redistribute the gradients can be a performance bottleneck and this leads to the idea of asynchronous training. For example, in the *Hogwild!* algorithm (Niu et al. 2011) the gradient from a node is used to update a central model whenever it is ready. The updated model is then redistributed to the node. This means that each node may have a slightly different version of the model at any given time and so the gradient updates may be stale; however, it works well in practice. Other, decentralized schemes have also been developed. For example, in Zhang et al. (2016b), the individual nodes update one another in a ring structure.

Data parallelism methods still assume that the entire model can be held in the memory of a single node. *Pipeline model parallelism* stores different layers of the network on different nodes and so does not have this requirement. In a naïve implementation, the first node runs the forward pass for the batch on the first few layers and passes the result to the next node, which runs the forward pass on the next few layers and so on. In the backward pass, the gradients are updated in the opposite order. The obvious disadvantage of this approach is that each machine lies idle for most of the cycle. Various schemes revolving around each node processing micro-batches sequentially have been proposed to reduce this inefficiency (e.g., Huang et al. 2019, Narayanan et al. 2021a). Finally, in *tensor model parallelism*, computation at a single network layer is distributed across nodes (e.g., Shoeybi et al. 2020). A good overview of distributed training methods can be found in Narayanan et al. (2021b) who combine tensor, pipeline, and data parallelism to train a language model with one trillion parameters on 3072 GPUs.

Problems

Problem 7.1 A two-layer network with two hidden units in each layer can be defined as:

$$\begin{aligned} y &= \phi_0 + \phi_1 \mathrm{a}\left[\psi_{01} + \psi_{11}\mathrm{a}[\theta_{01} + \theta_{11}x] + \psi_{21}\mathrm{a}[\theta_{02} + \theta_{12}x]\right] \\ &\quad + \phi_2 \mathrm{a}[\psi_{02} + \psi_{12}\mathrm{a}[\theta_{01} + \theta_{11}x] + \psi_{22}\mathrm{a}[\theta_{02} + \theta_{12}x]], \end{aligned} \tag{7.21}$$

where the functions $a[\bullet]$ are ReLU functions Compute the derivatives of the output y with respect to each of the 15 parameters $\boldsymbol{\phi}_\bullet$, $\theta_{\bullet\bullet}$, and $\psi_{\bullet\bullet}$ directly (i.e., not using the backpropagation algorithm). The derivative of the ReLU function with respect to its input $\partial a[z]/\partial z$ is the indicator function $\mathbb{I}[z > 0]$, which returns one if the argument is greater than zero and zero otherwise (figure 7.3).

Problem 7.2 What size are each of the terms in equation 7.8?

Problem 7.3 Calculate the derivative $\partial l_i/\partial f[\mathbf{x}_i, \boldsymbol{\phi}]$ for the least squares loss function:

$$l_i = (y - \mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}])^2. \tag{7.22}$$

Problem 7.4 Calculate the derivative $\partial l_i/\partial f[\mathbf{x}_i, \boldsymbol{\phi}]$ for the binary classification loss function:

$$l_i = -\sum_{i=1}^{I}(1-y_i)\log\left[1-\text{sig}[\text{f}[\mathbf{x}_i;\boldsymbol{\phi}]]\right] + y_i\log\left[\text{sig}[\text{f}[\mathbf{x}_i;\boldsymbol{\phi}]]\right], \tag{7.23}$$

where the function sig[•] is the logistic sigmoid and is defined as:

$$\text{sig}[z] = \frac{1}{1+\exp[-z]}. \tag{7.24}$$

Problem 7.5 Show that:

$$\frac{\partial}{\partial \mathbf{h}}\left(\boldsymbol{\beta}+\boldsymbol{\Omega}\mathbf{h}\right) = \boldsymbol{\Omega}^T, \tag{7.25}$$

where $\partial/\partial\mathbf{h}$ is the vector $[\partial/\partial h_1, \partial/\partial h_1, \ldots, \partial/\partial h_D]^T$, $\mathbf{h}$ and $\boldsymbol{\beta}$ are $D\times 1$ vectors and $\boldsymbol{\Omega}$ is a $D\times D$ matrix. If you aren't familiar with matrix calculus, then re-write the expression $\boldsymbol{\beta}+\boldsymbol{\Omega}\mathbf{h}$ in terms of sums of the constituent elements (e.g., so $\mathbf{Ab} = \sum_j a_{ij}b_j$), take the derivatives as normal, and then rewrite in matrix form.

Problem 7.6 Consider the case where we use the logistic sigmoid (see equation 7.24) as an activation function so $h = \text{sig}[f]$. Compute the derivative $\partial h/\partial f$ for this activation function. What happens to the derivative when the input takes (i) a large positive value and (ii) a large negative value?

Problem 7.7 Consider using (i) the Heaviside function and (ii) the rectangular function as activation functions:

$$\text{Heaviside}[z] = \begin{cases} 0 & z<0 \\ 1 & z\geq 0 \end{cases}, \tag{7.26}$$

and

$$\text{rect}[z] = \begin{cases} 0 & z<0 \\ 1 & 0\leq z\leq 1 \\ 0 & z>1 \end{cases}. \tag{7.27}$$

Discuss why these functions are problematic for neural network training with gradient-based optimization methods.

Problem 7.8 Show that:

$$\frac{\partial}{\partial \boldsymbol{\Omega}_k}\left(\boldsymbol{\beta}_k+\boldsymbol{\Omega}_k\mathbf{h}_k\right)\frac{\partial l_i}{\partial \mathbf{f}_k} = \frac{\partial l_i}{\partial \mathbf{f}_k}\mathbf{h}_k^T, \tag{7.28}$$

where $\partial/\partial\boldsymbol{\Omega}$ is a matrix where element (i,j) is given by $\partial/\partial\Omega_{ij}$. The terms $\boldsymbol{\beta}_k$, $\boldsymbol{\Omega}_k$, and $\mathbf{h}_k$ have sizes $D_{k+1}\times 1$, $D_{k+1}\times D_k$, and $D_k\times 1$ respectively. The term $\partial l_i/\partial\mathbf{f}_k$ is also of size $D_k\times 1$. As in problem 7.5, the easiest way to show this is to compute each term $\partial/\partial\Omega_{ij}$ separately, and then re-write in matrix form.

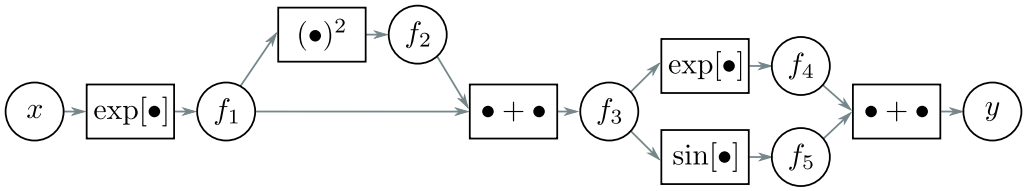

Figure 7.6 Computational graph for problems 7.12 and 7.13. Adapted from Domke (2010).

Problem 7.9 Write native Python code (i.e., *not* using PyTorch or Tensorflow) that implements the forward and backward passes of the backpropagation algorithm for a neural network with one input, one output, K hidden layers with D hidden units each, and a least squares loss function. Assume that there are 100 input examples drawn from a standard normal distribution, each of which maps to an output that is also drawn from a standard normal distribution, and that we are calculating the derivatives for the entire dataset (rather than a batch). Initialize the biases to all zeros and the weights from a standard normal distribution. How could you test whether the derivatives that you calculate are correct for a known input/output pair $\{x, y\}$?

Problem 7.10 Derive the equations for the backward pass of the backpropagation algorithm for a network that uses leaky ReLU activations, which are defined as:

$$\mathrm{a}[z] = \mathrm{ReLU}[z] = \begin{cases} \alpha z & z < 0 \\ z & z \geq 0 \end{cases}, \tag{7.29}$$

where α is a small positive constant (typically 0.1).

Problem 7.11 Consider the situation where we only have enough memory to store the values at every tenth hidden layer during the forward pass. Adapt the backpropagation algorithm to compute the derivatives in this situation using gradient checkpointing.

Problem 7.12 This problem explores computing derivatives on general acyclic computational graphs. Consider the function:

$$y = \exp\left[\exp[x] + \exp[x]^2\right] + \sin[\exp[x] + \exp[x]^2]. \tag{7.30}$$

We can break this down into a series of intermediate computations so that:

$$\begin{aligned} f_1 &= \exp[x] \\ f_2 &= f_1^2 \\ f_3 &= f_1 + f_2 \\ f_4 &= \exp[f_3] \\ f_5 &= \sin[f_3] \\ y &= f_4 + f_5. \end{aligned} \tag{7.31}$$

The associated computational graph is depicted in figure 7.6. Compute the derivative $\partial y/\partial x$, by *reverse mode differentiation*. In other words, compute in order:

$$\frac{\partial y}{\partial f_5}, \frac{\partial y}{\partial f_4}, \frac{\partial y}{\partial f_3}, \frac{\partial y}{\partial f_2}, \frac{\partial y}{\partial f_1} \text{ and } \frac{\partial y}{\partial x}, \tag{7.32}$$

using the chain rule in each case to make use of the derivatives already computed.

Problem 7.13 For the same function in problem 7.30, compute the derivative $\partial y/\partial x$, by *forward mode differentiation*. In other words, compute in order:

$$\frac{\partial f_1}{\partial x}, \frac{\partial f_2}{\partial x}, \frac{\partial f_3}{\partial x}, \frac{\partial f_4}{\partial x}, \frac{\partial f_5}{\partial x}, \text{ and } \frac{\partial y}{\partial x}, \tag{7.33}$$

using the chain rule in each case to make use of the derivatives already computed. Why do we not use forward mode differentiation when we compute the parameter gradients for deep networks?

Problem 7.14 Use your code from problem 7.9 to recreate figure 7.4 showing the phenomena of vanishing and exploding gradients.

Problem 7.15 Consider a random variable z_d with mean $\mathbb{E}[z_d] = 0$ and variance $\text{Var}[z_d] = \sigma^2$. Prove that if we pass this variable through the ReLU function:

$$z'_d = \text{ReLU}[z_d] = \begin{cases} 0 & z_d < 0 \\ z_d & z_d \geq 0 \end{cases}, \tag{7.34}$$

then the variance of the transformed variable is $\text{Var}[z'_d] = \sigma^2/2$.

Problem 7.16 Implement the code in figure 7.5 in PyTorch and plot the batch loss as a function of the number of iterations.

Problem 7.17 Change the code in figure 7.5 to tackle a binary classification problem. You will need to (i) change the targets y so that they are binary and (ii) change the loss function appropriately.

Chapter 8

Measuring performance

Previous chapters described neural network models, loss functions, and training algorithms. This chapter considers how to measure the performance of the trained models. With sufficient capacity (i.e., number of hidden units), a neural network model will often perform perfectly on the training data. However, this does not necessarily mean that it will generalize well to new test data.

We will see that the test errors have three distinct causes and that their relative contributions depend on (i) the inherent uncertainty in the task, (ii) the amount of training data, and (iii) the model capacity. The latter dependency raises the issue of hyperparameter search. We discuss how to select both the model hyperparameters (e.g., the number of hidden layers and the number of hidden units in each) and the learning algorithm hyperparameters (e.g., the learning rate and batch size).

8.1 Training a simple model

We explore model performance using the MNIST-1D dataset (figure 8.1). This consists of ten classes $y \in \{0, 1, \ldots 9\}$, representing the digits 0-9. The data are derived from 1D templates for each of the digits. Each data example $\mathbf{x}$ is created by choosing a template, randomly transforming this template, and finally adding noise. The full training dataset $\{\mathbf{x}_i, y_i\}$ consists of $I = 4000$ training examples, each of which consists of $D_i = 40$ dimensions representing the 40 sampled horizontal positions. The ten classes are drawn uniformly during data generation and so there are $\sim$400 examples of each class.

We use a neural network with $D_i = 40$ inputs and $D_o = 10$ outputs which are passed through the softmax function to produce class probabilities (see section 5.6). The network has two hidden layers with $D = 100$ hidden units each. It is trained with a multi-class cross-entropy loss using SGD with batch size 100 and learning rate 0.1 for 6000 steps (150 epochs). Figure 8.2 shows that the training error decreases as training proceeds. The training data are classified perfectly after about 4000 steps. The training loss also decreases, eventually approaching zero. Problem 8.1

However, this doesn't imply that the classifier is perfect; the model might have mem-

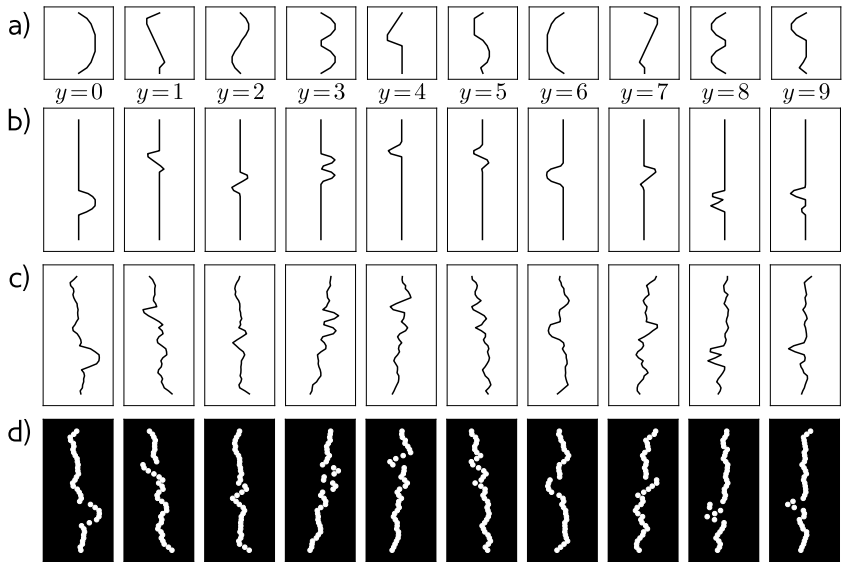

Figure 8.1 MNIST-1D (Greydanus 2020). a) Templates for 10 classes $y \in \{0, \ldots 9\}$, based on digits 0-9. b) Training examples $\mathbf{x}$ are created by randomly transforming a template and c) adding noise. d) The horizontal position of the transformed template is then sampled at 40 positions.

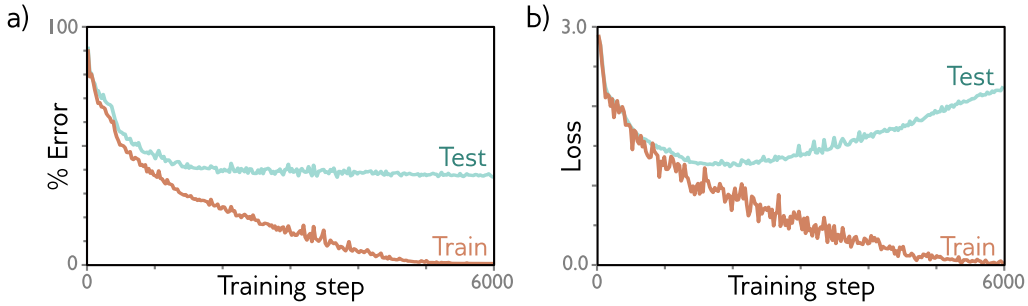

Figure 8.2 MNIST1D results. a) Percent classification error as a function of the training step. The training set errors decrease to zero, but the test errors do not decrease below $\sim 40\%$. This model doesn't generalize well to new test data. b) Loss as a function of the training step. The training loss decreases steadily towards zero. The test loss decreases at first but then increases as the model becomes increasingly confident about its (wrong) predictions.

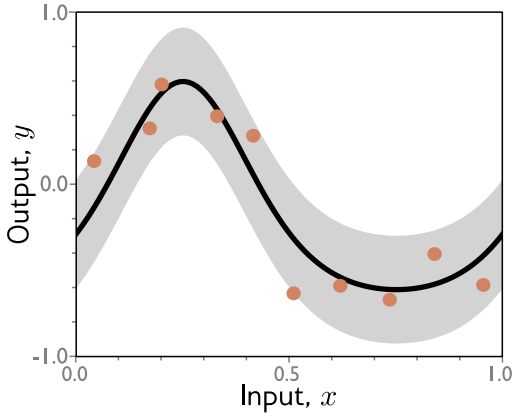

Figure 8.3 Regression function. Solid black line shows the ground truth function. To generate I training examples $\{x_i, y_i\}$, the input space $x \in [0, 1]$ is divided into I equal segments and one sample x_i is drawn rom a uniform distribution within each segment. The corresponding value y_i is created by evaluating the function at x_i and adding Gaussian noise (gray region shows ± 2 standard deviations). The test data are generated in the same way.

orized the training set but have no ability to predict new examples. To estimate the true performance, we need a separate *test set* of input/output pairs $\{\mathbf{x}_i, y_i\}$. To this end, we generate 1000 more examples using the same process. Figure 8.2a also shows the errors for this test data as a function of the training step. These decrease as training proceeds, but only to around 40%. This is better than the chance error rate of 90% error rate but far worse than for the training set; the model has not *generalized* well to the test data.

The test loss (figure 8.2b) decreases for the first 1500 training steps but then increases again. At this point, the test error rate is fairly constant; the model makes the same mistakes, but with increasing confidence. This decreases the probability of the correct answers, and thus increases the negative log-likelihood. This increasing confidence is a side-effect of the softmax function; the pre-softmax activations are driven to increasingly extreme values to make the probability of the training approach one (see figure 5.9).

8.2 Sources of error

We now consider the sources of the errors that occur when a model fails to generalize. To make this easier to visualize, we'll revert to a 1D linear least squares regression problem where we know exactly how the ground truth data were generated. Figure 8.3 shows a quasi-sinusoidal function; both training and test data are generated by sampling input values in the range $[0, 1]$, passing them through this function and adding Gaussian noise with a fixed standard deviation.

We'll fit a simplified shallow neural net to this data (figure 8.4). The weights and biases that connect the input layer to the hidden layer are chosen so that the 'joints' of the function are evenly spaced across the interval. If there are D hidden units, then these joints will be at $0, 1/D, 2/D, \ldots (D-1)/D$. This model can represent any piecewise linear function with D equally-sized regions in the range $[0, 1]$. As well as being easy to understand, this model also has the advantage that it can be fit in closed form without the need for stochastic optimization algorithms (see problem 8.3). Consequently, we can guarantee to find the global minimum of the loss function during training.

Problems 8.2-8.3

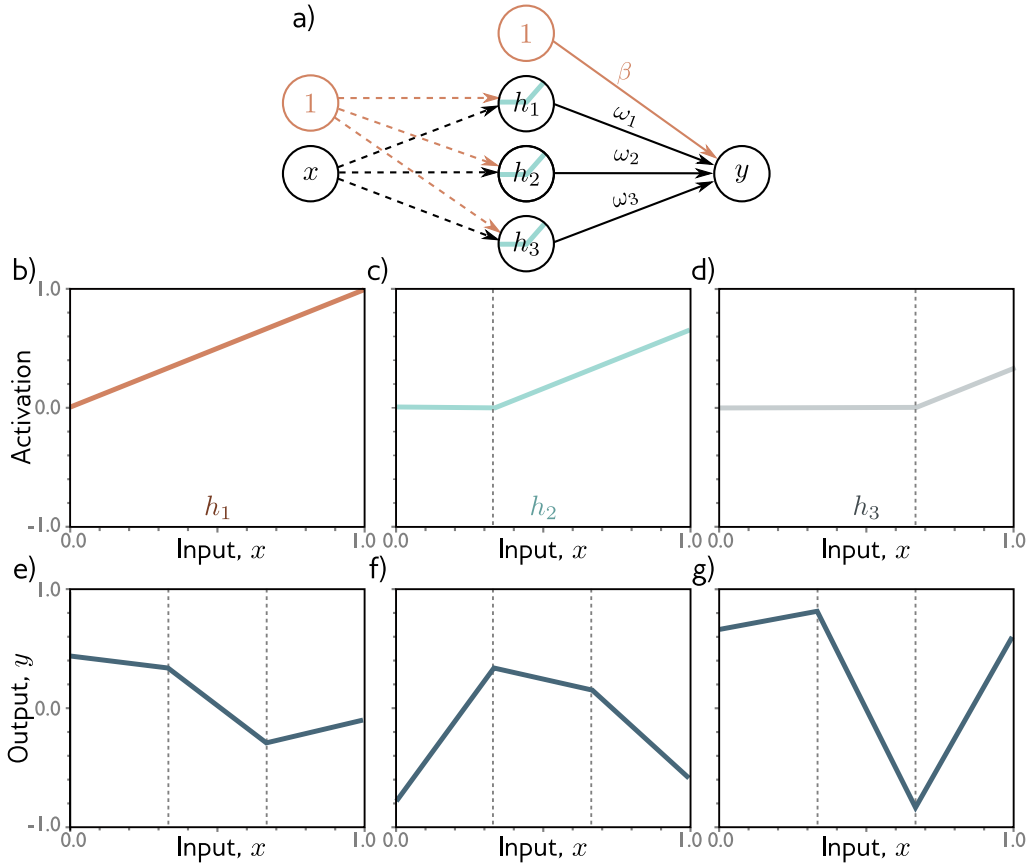

Figure 8.4 Simplified neural network with three hidden units. a) The weights and biases between the input and hidden layer are fixed (dashed arrows). b–d) They are chosen so that the hidden unit activations have slope one and their joints are equally spaced across the interval, with joints at $x = 0$, $x = 1/3$ and $x = 2/3$, respectively e–g) Modifying the remaining parameters $\boldsymbol{\phi} = \{\beta, \omega_1, \omega_2, \omega_3\}$ can create any piecewise linear function over $x \in [0, 1]$ with joints at 1/3 and 2/3.

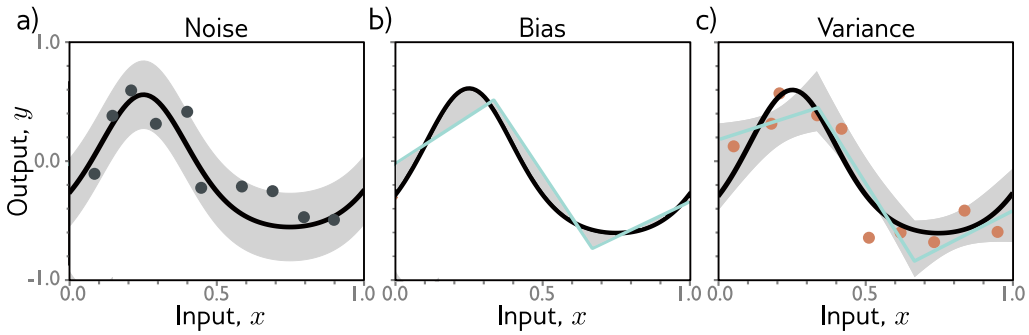

Figure 8.5 Sources of test error. a) Noise. Data generation is noisy, so even if the model exactly replicates the true underlying function (black line), the noise in the test data (gray points) means that some error will remain (gray region represents two standard deviations). b) Bias. Even with the best possible parameters, the three-region model (brown line) cannot fit the true function (black line) exactly. This bias is another source of error (gray regions represent signed error). c) Variance. In practice, we have limited noisy training data (orange points). When we fit the model, we don't recover the best possible function from panel (b) but a slightly different function (cyan line) that reflects idiosyncrasies of the training data. This provides a further source of error (gray region represents two standard deviations). Figure 8.6 shows how this region was calculated.

8.2.1 Noise, bias, and variance

There are three possible sources of error, which are known as *noise*, *bias*, and *variance* respectively (figure 8.5):

Noise The data generation process includes the addition of noise. This means that there are multiple possible valid outputs y for each input x (figure 8.5a). This source of error is insurmountable for the test data. Note that it does not necessarily limit the training performance; during training, it is likely that we never see the same input x twice, so it is still possible fit the training data perfectly.

Noise may arise because there is a genuine stochastic element to the data generation process, because some of the data are mislabeled, or because there are further explanatory variables that were not observed. In rare cases, noise may be absent; for example, a network might be used to approximate a function that is deterministic but requires significant computation to evaluate. However, noise is usually a fundamental limitation on the possible test performance.

Bias A second potential source of error may occur because the model is not flexible enough to fit the true function perfectly. For example, the three-region neural network model cannot exactly describe the quasi-sinusoidal function, even when the parameters are chosen optimally (figure 8.5b). This is known as *bias*.

Variance We have limited training examples and there is no way to distinguish systematic changes in the underlying function from noise. When we fit a model, we do not get the closest possible approximation to the true underlying function. Indeed, for different training datasets, the result will be slightly different each time. This additional source of variability is termed *variance* (figure 8.5c). In practice, there might also be additional variance due to the stochastic learning algorithm, which does not necessarily converge to the same solution each time.

8.2.2 Mathematical formulation of test error

We now make the notions of noise, bias, and variance mathematically precise. Consider a 1D regression problem where the data generation process has additive noise with variance σ^2 (e.g., figure 8.3); we can observe different outputs y for the same input x and so for each x there is a distribution $Pr(y|x)$ with mean $\mu[x]$:

Appendix B.2 Expectation

$$\mu[x] = \mathbb{E}_y[y[x]] = \int y[x] Pr(y|x) dy, \tag{8.1}$$

and fixed noise $\sigma^2 = \mathbb{E}_y\left[(\mu[x] - y[x])^2\right]$. Here we have used the notation $y[x]$ to specify that we are considering the output y at a given input position x.

Now consider a least squares loss between the model prediction $\mathrm{f}[x, \boldsymbol{\phi}]$ at position x and the observed value $y[x]$ at that position:

$$\begin{aligned} L[x] &= \big(\mathrm{f}[x, \boldsymbol{\phi}] - y[x]\big)^2 \\ &= \Big(\big(\mathrm{f}[x, \boldsymbol{\phi}] - \mu[x]\big) + \big(\mu[x] - y[x]\big)\Big)^2 \\ &= \big(\mathrm{f}[x, \boldsymbol{\phi}] - \mu[x]\big)^2 + 2\big(\mathrm{f}[x, \boldsymbol{\phi}]) - \mu[x]\big)\big(\mu[x] - y[x]\big) + \big(\mu[x] - y[x]\big)^2, \end{aligned} \tag{8.2}$$

where we have both added and subtracted the mean $\mu[x]$ of the underlying function in the second line and have expanded out the squared term in the third line.

Appendix B.2.1 Expectation rules

The underlying function is stochastic, and so this loss depends on the particular $y[x]$ we observe. The expected loss is:

$$\begin{aligned} &\mathbb{E}_y\big[L[x]\big] \\ &\quad= \mathbb{E}_y\Big[\big(\mathrm{f}[x, \boldsymbol{\phi}] - \mu[x]\big)^2 + 2\big(\mathrm{f}[x, \boldsymbol{\phi}] - \mu[x]\big)\big(\mu[x] - y[x]\big) + \big(\mu[x] - y[x]\big)^2\Big] \\ &\quad= \big(\mathrm{f}[x, \boldsymbol{\phi}] - \mu[x]\big)^2 + 2\big(\mathrm{f}[x, \boldsymbol{\phi}] - \mu[x]\big)\big(\mu[x] - \mathbb{E}_y\left[y[x]\right]\big) + \mathbb{E}_y\left[(\mu[x] - y[x])^2\right] \\ &\quad= \big(\mathrm{f}[x, \boldsymbol{\phi}] - \mu[x]\big)^2 + 2\big(\mathrm{f}[x, \boldsymbol{\phi}] - \mu[x]\big) \cdot 0 + \mathbb{E}_y\left[\big(\mu[x] - y[x]\big)^2\right] \\ &\quad= \big(\mathrm{f}[x, \boldsymbol{\phi}] - \mu[x]\big)^2 + \sigma^2, \end{aligned} \tag{8.3}$$

where in the second line, we have distributed the expectation operator and removed it from terms with no dependence on $y[x]$, and in the third line, we note that the second

term is zero, since $\mathbb{E}_y[y[x]] = \mu[x]$ by definition. Finally, in the fourth line, we have substituted in the definition of the noise σ^2. We can see that the expected loss has been broken down into two terms; the first term is the squared deviation between the model and the true function mean, and the second term is the noise.

The first term can be further partitioned into bias and variance. The parameters $\boldsymbol{\phi}$ of the model $\mathrm{f}[x, \boldsymbol{\phi}]$ depend on the training dataset $\mathcal{D} = \{x_i, y_i\}$ and so really we should write $\mathrm{f}[x, \boldsymbol{\phi}[\mathcal{D}]]$. The training dataset is a random sample from the data generation process; with a different sample of training data, we would learn different parameters. The expected model output $f_\mu[x]$ with respect to all possible datasets $\mathcal{D}$ is hence:

$$f_\mu[x] = \mathbb{E}_{\mathcal{D}}\big[\mathrm{f}[x, \boldsymbol{\phi}[\mathcal{D}]]\big]. \tag{8.4}$$

Returning to the first term of equation 8.3, we add and subtract $\mathrm{f}_\mu[x]$ and expand:

$$\begin{aligned}
&\big(\mathrm{f}[x, \boldsymbol{\phi}[\mathcal{D}]] - \mu[x]\big)^2 \qquad\qquad (8.5)\\
&\quad = \Big(\big(\mathrm{f}[x, \boldsymbol{\phi}[\mathcal{D}]] - f_\mu[x]\big) + \big(f_\mu[x] - \mu[x]\big)\Big)^2 \\
&\quad = \big(\mathrm{f}[x, \boldsymbol{\phi}[\mathcal{D}]] - f_\mu[x]\big)^2 + 2\big(\mathrm{f}[x, \boldsymbol{\phi}[\mathcal{D}]] - f_\mu[x]\big)\big(f_\mu[x] - \mu[x]\big) + \big(f_\mu[x] - \mu[x]\big)^2.
\end{aligned}$$

We then take the expectation with respect to the training data set $\mathcal{D}$:

$$\mathbb{E}_{\mathcal{D}}\Big[\big(\mathrm{f}[x, \boldsymbol{\phi}[\mathcal{D}]] - \mu[x]\big)^2\Big] = \mathbb{E}_{\mathcal{D}}\Big[\big(\mathrm{f}[x, \boldsymbol{\phi}[\mathcal{D}]] - f_\mu[x]\big)^2\Big] + \big(f_\mu[x] - \mu[x]\big)^2, \tag{8.6}$$

where we have simplified using similar steps as for equation 8.3. Finally, we substitute this result into equation 8.3:

$$\mathbb{E}_{\mathcal{D}}\Big[\mathbb{E}_y[L[x]]\Big] = \underbrace{\mathbb{E}_{\mathcal{D}}\Big[\big(\mathrm{f}[x, \boldsymbol{\phi}[\mathcal{D}]] - f_\mu[x]\big)^2\Big]}_{\text{variance}} + \underbrace{\big(f_\mu[x] - \mu[x]\big)^2}_{\text{bias}} + \underbrace{\sigma^2}_{\text{noise}}. \tag{8.7}$$

This equation says that the expected loss after taking into account the uncertainty in the training data $\mathcal{D}$ and the test data y consists of three components that are summed. The variance is uncertainty in the fitted model due to the particular training dataset we sample. The bias is the systematic deviation of the model from the mean of the function that we are modeling. The noise is the inherent uncertainty in the true mapping from input to output. These three sources of error will be present for any task. They combine additively for linear regression with a least squares loss. However, their interaction can be more complex for other types of problem.

8.3 Reducing error

In the previous section, we saw that test error results from three sources: noise, bias, and variance. The noise component is insurmountable; there is nothing we can do to circumvent this, and it represents a fundamental limit on model performance. However, it is possible to reduce the other two terms.

8.3.1 Reducing variance

Recall that the variance results from limited noisy training data. Fitting the model to two different training sets results in slightly different parameters. It follows that we can reduce the variance by increasing the quantity of training data. This averages out the inherent noise and ensures that the input space is well sampled.

Figure 8.6 shows the effect of training with 6, 10, and 100 samples. For each dataset size, we show the best fitting model for three training datasets. With only six samples, the fitted function is quite different each time; the variance is large. As we increase the number of samples, the fitted models become very similar and the variance reduces. In general, adding training data almost always improves test performance.

8.3.2 Reducing bias

The bias term results from the inability of the model to describe the true underlying function. This suggests that we can reduce this error by making the model more flexible. This is usually done by increasing the model *capacity*, which for neural networks means adding more hidden units and/or hidden layers.

In the simplified model, adding capacity corresponds to adding more hidden units in such a way that the interval $[0, 1]$ is divided into more linear regions. Figures 8.7a-c show that (unsurprisingly) this does indeed reduce the bias; as we increase the number of linear regions to ten, the model becomes flexible enough to fit the true function closely.

8.3.3 Bias-variance trade-off

However, figures 8.7d-f show an unexpected side-effect of increasing the model capacity. For a fixed-size training dataset, the variance term increases as the model capacity increases. Consequently, increasing the model capacity does not necessarily reduce the test error. This is known as the *bias-variance trade-off*.

Figure 8.8 explores this phenomenon. In panels a-c) we fit the simplified model with three linear regions to three different datasets of 15 points. Although the datasets differ, the final model is much the same; the noise in the dataset roughly averages out in each linear region. In panels d-f) we fit a model with ten regions to the same three datasets. This model has more flexibility, but this is disadvantageous; the model certainly fits the data better and the training error will be lower, but much of the extra descriptive power is devoted to modeling the noise. This phenomenon is known as *overfitting*.

We've seen that for a fixed-size training dataset, the bias will decrease, but the variance will increase as we add capacity to the model. This suggests that there is an optimal capacity where the bias is not too large, and the variance is still relatively small. Figure 8.9 shows how these terms vary numerically for the toy model as we increase the capacity, using the data from figure 8.8. For regression models, the total expected error is the sum of the bias and the variance, and this sum is minimized when the model capacity is four (i.e., it has four hidden units and four linear regions).

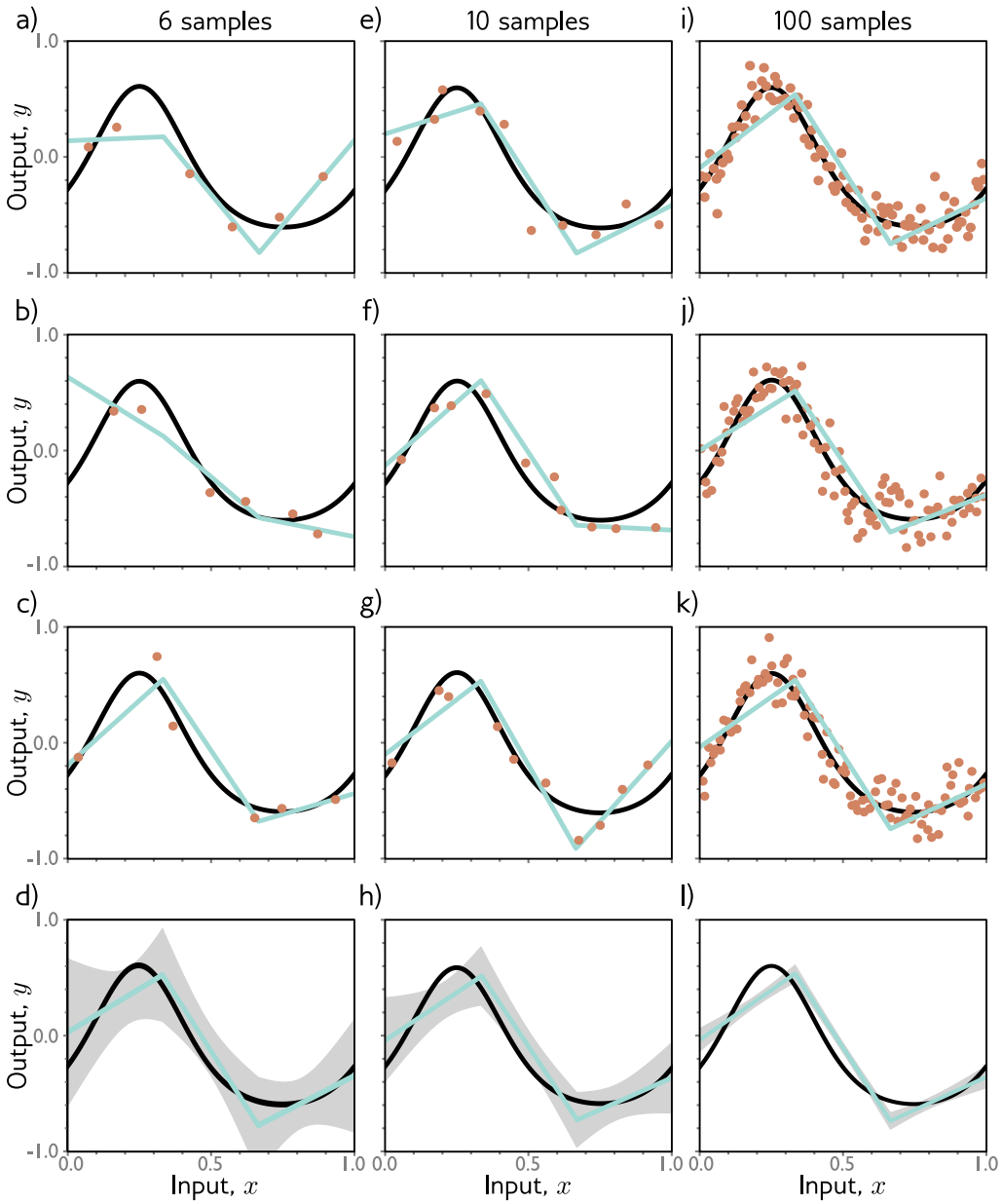

Figure 8.6 Reducing variance by increasing training data. a-c) The three-region model fitted to three different randomly sampled datasets of six points. The fitted model is quite different each time. d) We repeat this experiment many times and plot the mean model predictions (cyan line) and the variance of the model predictions (gray area shows two standard deviations). e-h) We do the same experiment, but this time with a dataset of size ten. The variance of the predictions is reduced. i-l) We repeat this experiment with datasets of size 100. Now the fitted model is always similar, and the variance is small.

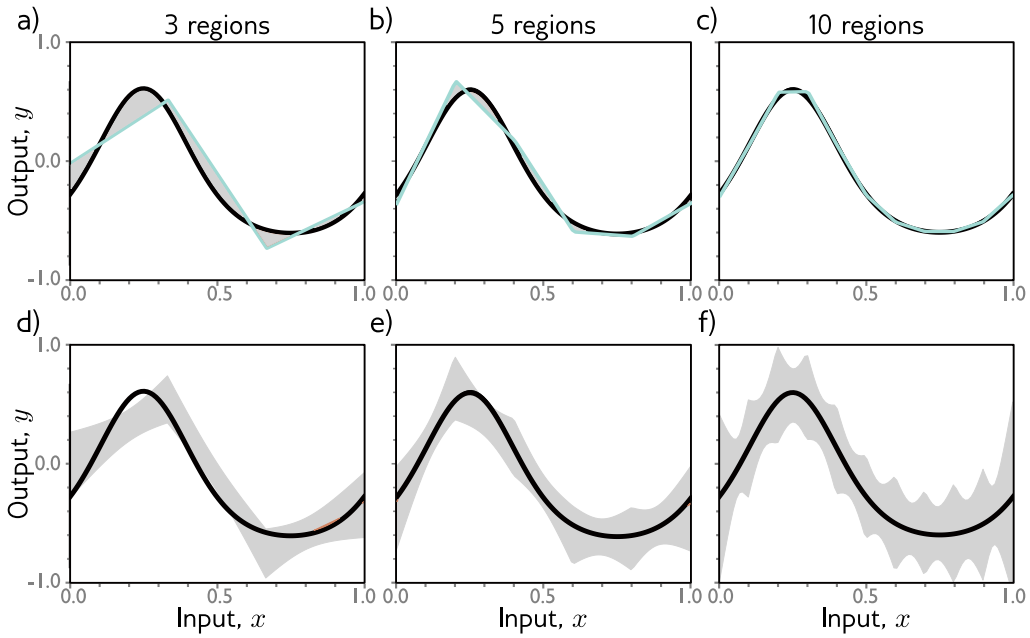

Figure 8.7 Bias and variance as a function of model capacity. a-c) As we increase the number of hidden units of the toy model, the number of linear regions increases, and the model becomes able to fit the true function closely; the bias (gray region) decreases. d-f) Unfortunately, increasing the capacity of the model has the side-effect of increasing the variance term (gray region). This is known as the bias-variance trade-off.

8.4 Double descent

In the previous section we examined the bias-variance trade-off as we increase the capacity of a model. Let's now return to the MNIST1D dataset and see whether this happens in practice. We use 10,000 training examples, test with another 5,000 examples, and examine the training and test performance as we increase the capacity (number of parameters) in the model. We train the model with ADAM and a step size of 0.005 using the full batch of 10,000 examples for 4000 steps.

Figure 8.10a shows the training and test error for a neural network with two hidden layers as the number of hidden units is increased. The training error decreases as the capacity grows and becomes close to zero very quickly. The vertical dashed line represents the capacity where the model has the same number of parameters as there are training examples, but the model memorizes the dataset before this point. The test error decreases as we add model capacity but does not increase as predicted by the bias-variance trade-off curve; it just keeps decreasing.

In figure 8.10b we repeat this experiment, but this time, we randomize 15% of the

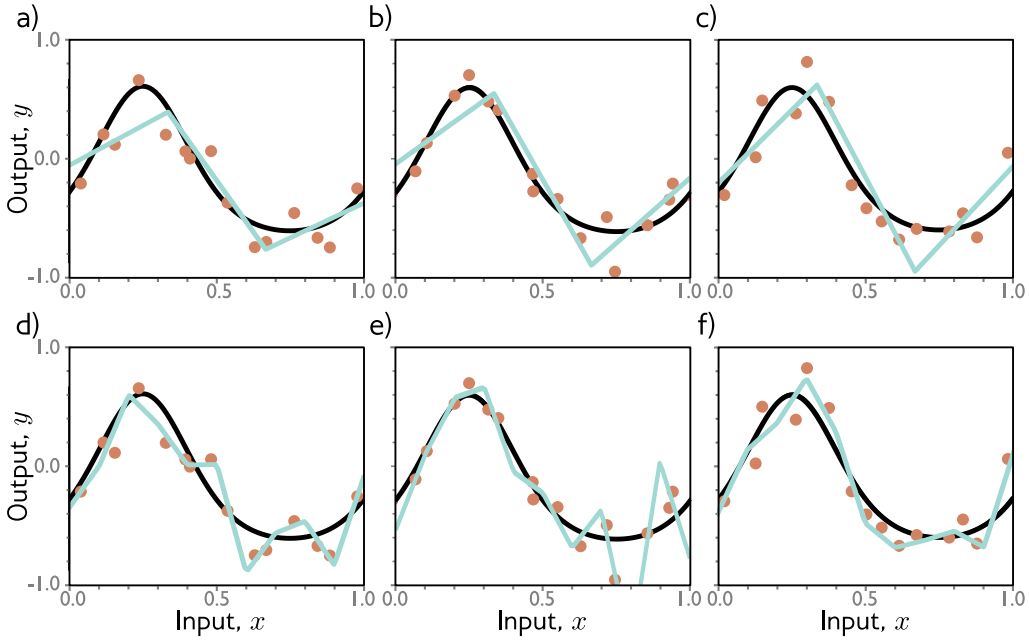

Figure 8.8 Overfitting. a-c) A model with three regions is fit to three different datasets of fifteen points each. The result is very similar in all three cases (i.e., the variance is low). d-f) A model with ten regions is fit to the same datasets. The additional flexibility does not necessarily produce better predictions. While these three models each describe the training data better, they are not necessarily closer to the true underlying function (black curve). Instead, they overfit the data and describe the noise, and the variance (difference between fitted curves) is larger.

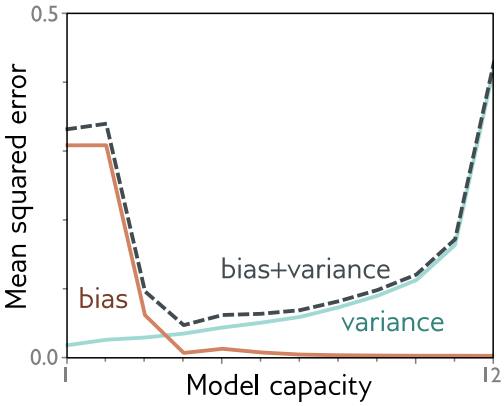

Figure 8.9 Bias-variance trade-off. The bias and variance terms from equation 8.7 are plotted as a function of the model capacity (number of hidden units / linear regions) in the simplified model using training data from figure 8.8. As the capacity increases, the bias (solid orange line) decreases but the variance (solid cyan line) increases. The sum of these two terms (dashed gray line) is minimized when the capacity is four.

training labels. Once more, the training error decreases to zero. This time, there is more randomness and the model requires almost as many parameters as there are data points to memorize the data. The test error does show the typical bias-variance trade-off as we increase the capacity to the point where the training data is fitted exactly. However, then it does something unexpected; it starts to decrease again. Indeed, if we add enough capacity, the test loss reduces to below the minimal level that we achieved in the first part of the curve.

This phenomenon is known as *double descent*. For some datasets like MNIST it is present with the original data (figure 8.10c). For others, like MNIST1D and CIFAR (figure 8.10d), it emerges or becomes more prominent when we add noise to the labels. The first part of the curve is referred to as the *classical* or *under-parameterized regime* and the second part as the *modern* or *over-parameterized regime*. The central part where the error increases is termed the *critical regime*.

8.4.1 Explanation

The discovery of double descent is recent, unexpected, and somewhat puzzling. What we are seeing results from an interaction of two phenomena. First, the test performance becomes temporarily worse when the model has just enough capacity to memorize the data. Second, the test performance continues to improve with capacity even after the training performance is perfect. The first phenomenon is exactly as predicted by the bias-variance trade-off. The second phenomenon is more confusing; it's not clear why performance should be better in the over-parameterized regime given that there are now not even enough training data points to uniquely constrain the model parameters.

To understand why performance continues to improve as we add more parameters, note that once the model has enough capacity to drive the training loss to near zero, the model fits the training data almost perfectly. This implies that further capacity cannot help the model fit the training data any better; any change must be occurring *between* the training points. The tendency of a model to prioritize one solution over another as it extrapolates between data points is known as its *inductive bias*.

Problems 8.4-8.5

The behavior of the model between data points is particularly important because in high-dimensional space, the training data are extremely sparse. The MNIST-1D dataset has 40 dimensions and we trained with 10,000 examples. If this seems like plenty of data, consider what would happen if we were to quantize each dimension of the input into 10 bins. There would be 10^{40} bins in total, and these are constrained by only 10^5 examples. Even with this coarse quantization, there will only be one data point in every 10^{35} bins! The tendency of the volume of high-dimensional space to overwhelm the number of training points is termed the *curse of dimensionality*.

The implication is that problems in high dimensions might look more like figure 8.11a; there are small regions of the input space where we observe data with large gaps between them. The putative explanation for double descent is that as we add capacity to the model, it interpolates between the nearest data points increasingly smoothly. In the absence of information about what happens between the training points, assuming smoothness is a sensible strategy, which will probably generalize reasonably to new data.

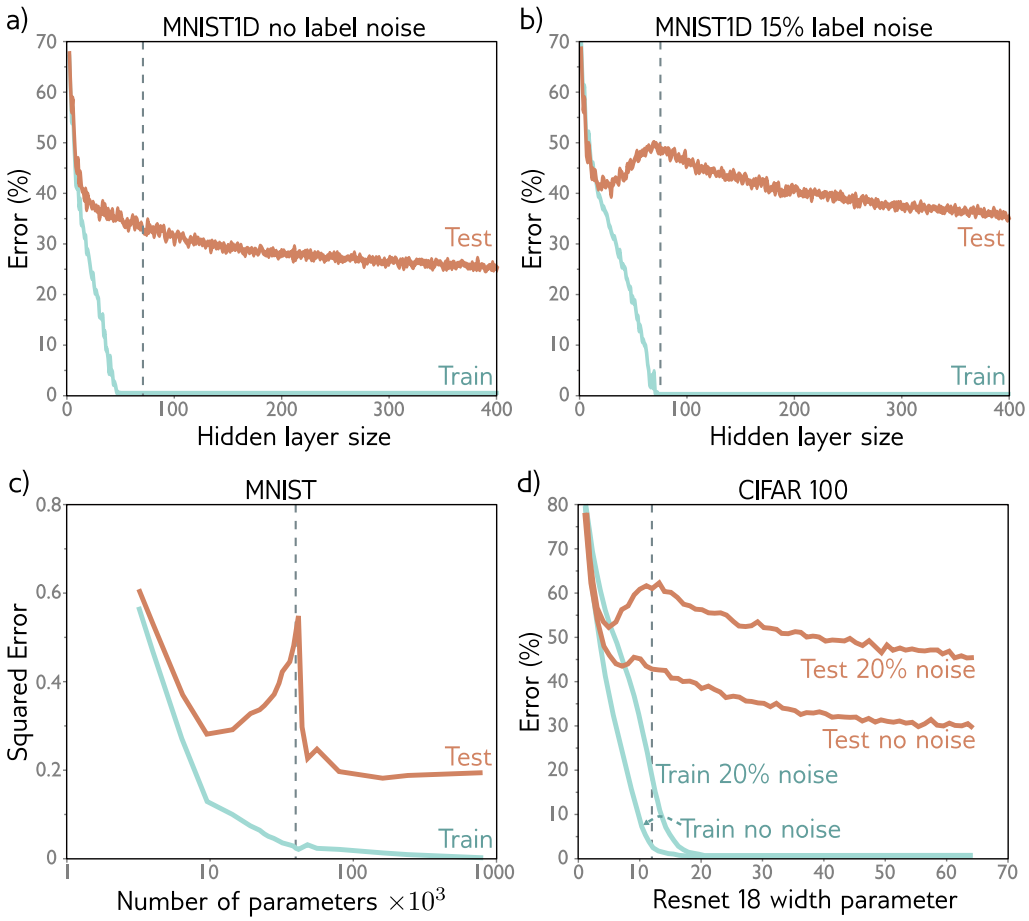

Figure 8.10 Double descent. a) Training and test loss on MNIST1D for a two hidden layer network as we increase the number of hidden units (and hence parameters) in each layer. The training loss decreases to zero as the number of parameters approaches the number of training examples (vertical dashed line). The test error does not show the expected bias-variance trade-off but continues to decrease even after the model has memorized the dataset. b) The same experiment is repeated with noisier training data. Again, the training error reduces to zero, although it now takes almost as many parameters as there are training points to memorize the dataset. The test error shows the predicted bias/variance trade-off; it decreases as the capacity increases, but then increases again as we near the point where the training data is exactly memorized. However, it then subsequently decreases again and ultimately reaches a better performance level. This is known as double descent. Depending on the loss, the model, and the amount of noise in the data, the double descent pattern can be seen to a greater or lesser degree across many datasets. c) Results on MNIST (without label noise) with shallow neural network from Belkin et al. (2019). d) Results on CIFAR 100 with ResNet18 network (see chapter 11) from Nakkiran et al. (2019). See original papers for experimental details.

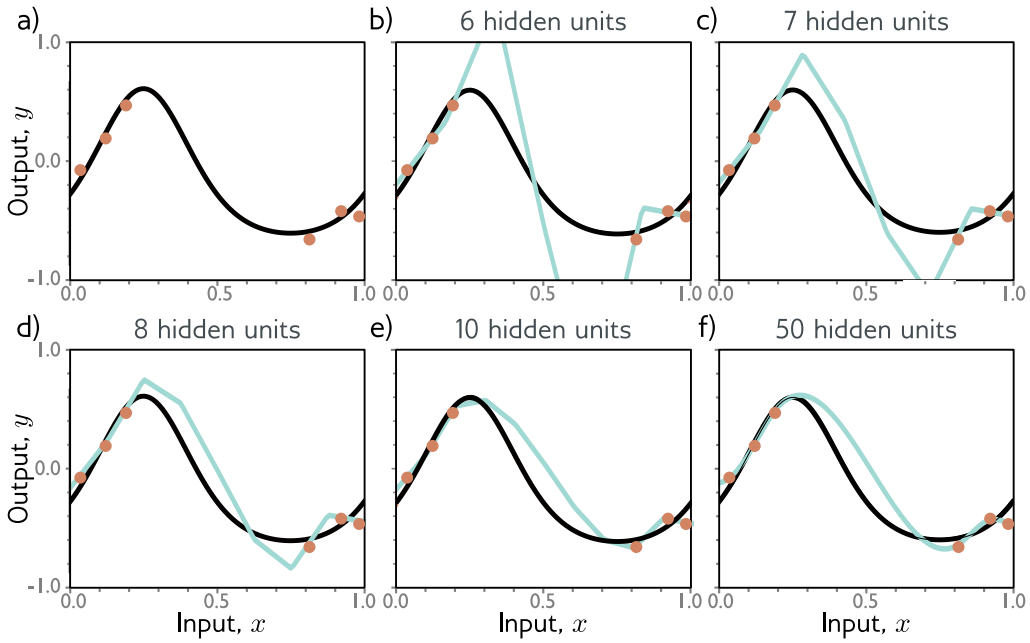

Figure 8.11 Increasing capacity allows smoother interpolation between sparse data points. a) Consider this situation where the training data (orange circles) are sparse and there is a large region in the center with no data examples to help the model fit the true function (black curve). b) If we fit a model that has just enough capacity to fit the training data (cyan curve), then it will have to contort itself to pass through the training data, and the output predictions will not be smooth. c-f) However, as we add more hidden units, the model has the *ability* to interpolate between the points more smoothly (smoothest possible curve plotted in each case). However, unlike here, it is not obliged to.

This argument is plausible. It's certainly true that as we add more capacity to the model, it will have the capability to create smoother functions. Figures 8.11b-f show the smoothest possible functions that still pass through the data points as we increase the number of hidden units. When the number of parameters is very close to the number of training data examples (figure 8.11), the model is forced to contort itself to fit exactly, resulting in erratic predictions. This explains why the peak in the double descent curve is so pronounced. As we add more hidden units, the model has the ability to construct smoother functions that are likely to generalize better to new data.

However, this does not explain *why* over-parameterized models should produce smooth functions. Figure 8.12 shows three functions that can equally be created by the simplified model with 50 hidden units. In each case, the model fits the data exactly, and so the loss is zero. If the modern regime of double descent is explained by increasing smoothness, then what exactly is encouraging this smoothness?

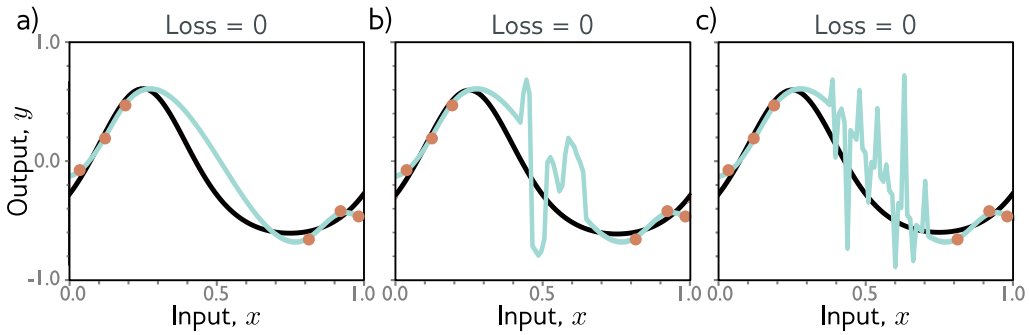

Figure 8.12 Regularization. a-c) Each of the three fitted curves passes through the data points exactly and so the training loss for each is zero. However, we might expect the smooth curve in panel (a) to generalize much better to new data than the erratic curves in panels (b) and (c). Any factor that biases a model towards a subset of the solutions with a similar training loss is known as a regularizer. It is thought that the initialization and/or fitting of neural networks has an implicit regularizing effect. Consequently, in the over-parameterized regime, more reasonable solutions such as that in panel (a) are encouraged.

There are two potential answers to this question. First, the network initialization may encourage smoothness and during fitting the model never departs from the sub-domain of smooth functions. Second, the training algorithm may somehow "prefer" to converge to smooth functions. Any factor that biases a solution towards a subset of equivalent solutions is known as a *regularizer*, and so one possibility is that the training algorithm acts as an implicit regularizer. We return to this topic in chapter 9.

8.5 Choosing hyperparameters

In the previous section, we discussed how test performance changes with model capacity. Unfortunately, in the classical regime we don't have access to either the bias (which requires knowledge of the true underlying function) or the variance (which requires multiple independently sampled datasets to estimate). In the modern regime, there is no way to tell how much capacity should be added before the test error stops improving. This raises the question of exactly how we should choose the capacity in practice.

For a deep network, the model capacity depends on the numbers of hidden layers and hidden units per layer as well as other aspects of architecture that we have yet to introduce. Furthermore, the choice of learning algorithm and any associated parameters (learning rate, etc.) also affects the test performance. These elements are collectively termed the hyperparameters. The process of finding the best hyperparameters is known as *hyperparameter search*.

Hyperparameters are typically chosen empirically; we train many models with different hyperparameters on the same training set, measure their performance, and retain the best model. However, we do not measure their performance on the test set; this would admit the possibility that these hyperparameters just happen to work well for the test set but don't generalize to further data. Instead, we introduce a third dataset known as a *validation set*. For every choice of hyperparameters, we train the associated model using the training set and evaluate performance on the validation set. Finally, we select the model that worked best on the validation set and measure its performance on the test set. In principle, this should give a good estimate of the true performance.

The hyperparameter space is generally smaller than the parameter space, but still too large to try every combination exhaustively. Unfortunately, many hyperparameters are discrete (e.g., the number of hidden layers) and others may be conditional on one another (e.g., we only need to specify the number of hidden units in the tenth hidden layer if there are ten or more layers). Hence, we cannot rely on gradient descent methods as we did for learning the model parameters. Hyperparameter optimization algorithms intelligently sample the space of hyperparameters, contingent on previous results. This procedure is computationally expensive since an entire model must be trained and the validation performance measured for each combination of hyperparameters.

8.6 Summary

To measure performance, we use a separate test set. The degree to which performance is maintained on this test set is known as generalization. Test errors can be explained by three factors: noise, bias, and variance. These combine additively in regression problems with least squares losses. Adding training data decreases the variance. When the model capacity is less than the number of training examples, increasing the capacity decreases bias, but increases variance. This is known as the bias-variance trade-off, and there is a capacity where the trade-off is optimal.

However, this is balanced against a tendency for performance to improve with capacity, even when the parameters exceed the training examples. Together, these two phenomena create the double descent curve. It is thought that the model interpolates more smoothly between the training data points in the over-parameterized "modern regime" although it is not clear what drives this. To choose the capacity and other model and training algorithm hyperparameters, we fit multiple models and evaluate their performance using a separate validation set.

Notes

Bias-variance trade-off: We showed that the test error for regression problems with least squares loss decomposes into the sum of noise, bias, and variance terms. These factors are all present for models with other losses, but their interaction is typically more complicated (Friedman 1997; Domingos 2000). For classification problems there are some counterintuitive

predictions; for example, if the model is biased towards selecting the wrong class in a region of the input space, then increasing the variance can improve the classification rate as this pushes some of the predictions over the threshold to be classified correctly.

Cross-validation: We saw that it is typical to divide the data into three parts: training data (which is used to learn the model parameters), validation data (which is used to choose the hyperparameters), and test data (which is used to estimate the final performance). This approach is known as *cross-validation*. However, this division may cause problems in situations where the total number of data examples is limited; as we saw earlier in this chapter, if the number of training examples is comparable to the model capacity, then the variance will be large.

One way to reduce this problem is to use *k-fold cross-validation*. The training and validation data are partitioned into K disjoint subsets. For example, we might divide these data into five parts. We train with four and validate with the fifth for each of the five permutations and choose the hyperparameters based on the average validation performance. The final test performance is assessed by using the average of the predictions from the five models with the best hyperparameters on a completely different test set. There are many variations of this idea, but all share the general goal of using a larger proportion of the data to train the model and hence reduce variance.

Capacity: We have used the term *capacity* informally to mean the number of parameters or hidden units in the model (and hence indirectly, the ability of the model to fit functions of increasing complexity). The *representational capacity* of a model describes the space of possible functions it can construct when we consider all possible parameter values. When we take into account the fact that an optimization algorithm may not be able to reach all of these solutions, what is left is the *effective capacity*.

The Vapnik-Chervonenkis (VC) dimension (Vapnik & Chervonenkis 1971) is a more formal measure of capacity. It is the largest number of training examples that a binary classifier can label arbitrarily. Bartlett et al. (2017b) derive upper and lower bounds for the VC-dimension in terms of the number of layers and weights. An alternative measure of capacity is the Rademacher complexity, which is the expected empirical performance of a classification model (with optimal parameters) for data with random labels. Neyshabur et al. (2017a) derive a lower bound on the generalization error in terms of the Rademacher complexity.

Double descent: The term "double descent" was coined by Belkin et al. (2019) who demonstrated that the test error decreases again in the over-parameterized regime for two-layer neural networks and random features. They also claimed that this occurs in decision trees although Buschjäger & Morik (2021) subsequently provided evidence to the contrary. Nakkiran et al. (2019) show that double descent occurs for various modern datasets (CIFAR-10, CIFAR-100, IWSLT'14 de-en), architectures (CNNs, ResNets, transformers), and optimizers (SGD, Adam). The phenomenon is more pronounced when noise is added to the target labels (Nakkiran et al. 2019) and also when some regularization techniques are used (Ishida et al. 2020).

Nakkiran et al. (2019) also provide empirical evidence that test performance is dependent on *effective model capacity*, which is the largest number of samples for which a given model and training method can achieve zero training error. This is the point at which the model starts to devote its efforts to interpolating smoothly. As such, the test performance depends not just on the model, but also the training algorithm and length of training. They observe the same pattern when they study a model with fixed capacity and increase the number of training iterations. They term this *epoch-wise double descent*.

Double descent makes the rather strange prediction that adding training data can sometimes make test performance worse. Consider an over-parameterized model that is in the second descending part of the curve. If we increase the amount of training data so that it matches the

model capacity, we will now be in the critical region of the new test error curve and the test loss may increase.

Bubeck & Sellke (2021) prove that overparameterization is necessary if we want to interpolate data smoothly in high dimensions. They demonstrate that there is a trade-off between the number of parameters and the Lipschitz constant of a model (the fastest the output can change for a small change in the input). A review of the theory of over-parameterized machine learning can be found in Dar et al. (2021).

Curse of dimensionality: As dimensionality increases, the volume of space grows so fast that the amount of data needed to densely sample it increases exponentially. This phenomenon is known as the curse of dimensionality. In general, high-dimensional space has many unexpected properties, and caution should be used when trying to reason about it based on low-dimensional examples. This book visualizes many aspects of deep learning in one or two dimensions but these visualizations should be treated with healthy skepticism.

Problems 8.6-8.9

The following are some of the surprising properties of high-dimensional space: (i) Two randomly sampled data points from a standard normal distribution are very close to orthogonal to one another with high likelihood. (ii) The distance from the origin of samples from a standard normal distribution is roughly constant. (iii) Most of a volume of a high-dimensional sphere (hypersphere) is adjacent to its surface (a common metaphor is that most of the volume of a high-dimensional orange is in the peel, not in the pulp). (iv) If we place a unit diameter hypersphere inside a hypercube with unit length sides, then the hypersphere takes up a decreasing proportion of the volume of the cube as the dimension increases. Since the volume of the cube is fixed at size one, this implies that the volume of a high-dimensional hypersphere becomes close to zero. (v) If we generate a set of random points in a high-dimensional hypercube, then the ratio of the Euclidean distance between the nearest and furthest points becomes close to one. For further information, consult Beyer et al. (1999) and Aggarwal et al. (2001).

Real-world performance: In this chapter, we argued that model performance can be evaluated by using a held-out test set. However, this is not necessarily indicative of real-world performance if the statistics of the test set do not match the statistics of data in the real world. Moreover, the statistics of real-world data may change over time, causing the model to become increasingly stale and performance to decrease. This is known as *data drift* and means that deployed models must be carefully monitored.

There are three main reasons why real-world performance may be worse than the test performance implies. First, the statistics of the input data $\mathbf{x}$ may change; we may then be observing parts of the function that were sparsely sampled or not sampled at all during training. This is known as *covariate shift*. Second, the statistics of the output data $\mathbf{y}$ may change; if some values are very rare during training, then the model may learn not to predict these in ambiguous situations and will make mistakes if they are more common in the real world. This is known as *prior shift*. Third, the relationship between input and output may change. This is known as *concept shift*. These issues are discussed in Moreno-Torres et al. (2012).

Hyperparameter search: Finding the best hyperparameters is a challenging optimization task. Testing a single configuration of hyperparameters is expensive; we have to train an entire model and measure its performance. We have no easy way to access the derivatives (i.e., how performance changes when we make a small change to a hyperparameter) and anyway many of the hyperparameters are discrete, so we cannot use gradient descent methods. There are multiple local minima and no way to tell if we are close to the global minimum. The noise level is high since each training/validation cycle uses a stochastic training algorithm; we don't expect to get the same results if we train a model twice with the same hyperparameters. Finally, some variables are conditional and only exist if others are set. For example, the number of hidden units in the third hidden layer of a network is only relevant if we have at least three hidden layers.

A simple approach is just to sample the space randomly (Bergstra & Bengio 2012). However, for continuous variables, a better method is to build a model of the underlying hyperparameter/performance function and the uncertainty in this function. This can be exploited to test where the uncertainty is great (explore the space), or home in on regions where performance looks promising (exploit previous knowledge). Bayesian optimization is a framework based on Gaussian processes that does just this and its application to hyperparameter search is described in Snoek et al. (2012). The Beta-Bernoulli bandit (see Lattimore & Szepesvári 2020) is a roughly equivalent model for describing uncertainty in results due to discrete variables.

The sequential model-based configuration (SMAC) algorithm (Hutter et al. 2011) can cope with continuous, discrete, and conditional parameters. The basic approach is to use a random forest to model the objective function where the mean of the tree predictions is the best guess about the objective function and their variance represents the uncertainty. A completely different approach that can also cope with combinations of continuous, discrete, and conditional parameters is Tree-Parzen Estimators (Bergstra et al. 2011). The previous methods modeled the probability of the model performance given the hyper-parameters. In contrast, the Tree-Parzen estimator models the probability of the hyper-parameters given the model performance.

Hyperband (Li et al. 2016b) is a multi-armed bandit strategy for hyperparameter optimization. It assumes that there are computationally cheap but approximate ways to measure performance (e.g., by not training to completion), and that these can be associated with a budget (e.g., by training for a fixed number of iterations). A number of random configurations are sampled and run until the budget is used up. Then the best fraction η of runs is kept and the budget is multiplied by $1/\eta$. This is repeated until the maximum budget is reached. This approach has the advantage of efficiency; for bad configurations, it does not need to run the experiment to the end. However, each sample is just chosen randomly which is inefficient. The BOHB algorithm (Falkner et al. 2018) combines the efficiency of Hyperband with the more sensible choice of parameters from Tree Parzen estimators to construct an even better method.

Problems

Problem 8.1 Will the multi-class cross-entropy training loss in figure 8.2 ever reach zero? Explain your reasoning.

Problem 8.2 What values should we choose for the three weights and the bias in the first layer of the model in figure 8.4a so that the responses at the hidden units are as depicted in figures 8.4b–d?

Problem 8.3 Given a training dataset consisting of I input/output pairs $\{x_i, y_i\}$, show how the parameters $\{\beta, \omega_1, \omega_2, \omega_3\}$ can be found in closed form for the model in figure 8.4a using the least squares loss function.

Problem 8.4 Consider the curve in figure 8.10b at the point where we train a model with a hidden layer of size 200, which would have 50,410 parameters. What do you predict will happen to the training and test performance if we increase the number of training examples from 10,000 to 50,410?

Problem 8.5 Consider the case where the model capacity exceeds the number of training data points, and the model is flexible enough to reduce the training loss to zero. What are the implications of this for fitting a heteroscedastic model? Propose a method to resolve any problems that you identify.

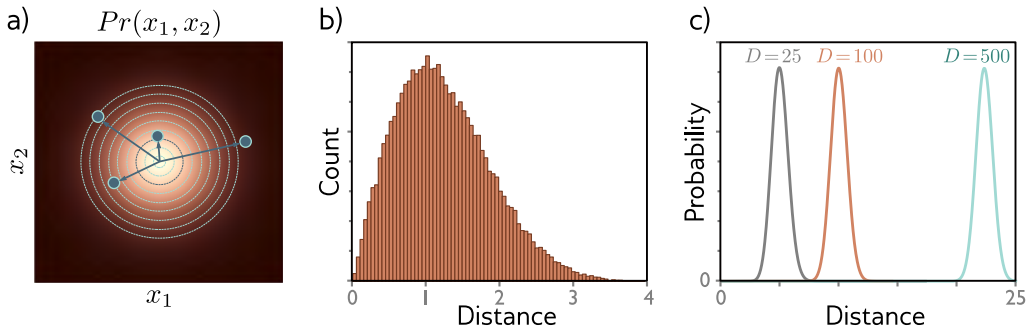

Figure 8.13 Typical sets. a) Standard normal distribution in two dimensions. Circles are four samples from this distribution. As the distance from the center increases, the probability decreases but the volume of space at that radius (i.e., the area between adjacent evenly spaced circles) increases. b) These factors trade off so that the histogram of distances of samples from the center has a pronounced peak. c) In higher dimensions, this effect becomes more extreme and the probability of observing a sample close to the mean becomes vanishingly small. Although the most likely point is at the mean of the distribution, the *typical samples* are found in a relatively narrow shell.

Problem 8.6 Prove that the angle between two random samples from a 1000–dimensional standard Gaussian distribution is close to zero with high probability.

Problem 8.7 The volume of a hypersphere with radius r in D dimensions is:

$$\text{Vol}[r] = \frac{r^D \pi^{D/2}}{\Gamma[D/2+1]}, \tag{8.8}$$

Appendix ?? Gamma function

where $\Gamma[\bullet]$ is the Gamma function. Show either mathematically or by writing code that the volume of a hypersphere of diameter one (radius $r = 0.5$) becomes zero as the dimension increases.

Problem 8.8 Consider a hypersphere of radius $r = 1$. Calculate the proportion of the total volume that is in the 1% of the radius that is closest to the surface of the hypersphere as a function of the dimension.

Problem 8.9 Figure 8.13c shows the distribution of distances of samples of a standard normal distribution as the dimension increases. Empirically verify this finding by sampling from the standard normal distributions in 25, 100, and 500 dimensions and plotting a histogram of the distances from the center. What closed-form probability distribution describes these distances?

Chapter 9

Regularization

Chapter 8 described how to measure model performance and identified that there can be a large performance gap between the training and test data. Possible reasons for this discrepancy include: (i) the model describes statistical peculiarities of the training data that are not representative of the true mapping from input to output (overfitting), and (ii) the model is unconstrained in areas where there are no training examples, and this leads to suboptimal predictions.

This chapter discusses *regularization* techniques. These are a family of methods that reduce the generalization gap between training and test performance. Strictly speaking, regularization involves adding explicit terms to the loss function that favor certain parts of parameter space. However, in machine learning, this term is often used to refer to any strategy that improves generalization.

We start by considering regularization in its strictest sense. Then we'll consider how the stochastic gradient descent algorithm itself favors certain solutions. This is known as implicit regularization. Following this, we'll consider a set of heuristic methods that improve test performance. These include early stopping, ensembling, dropout, label smoothing, and transfer learning.

9.1 Explicit regularization

Consider fitting a model $\mathrm{f}[\mathbf{x}, \boldsymbol{\phi}]$ with parameters $\boldsymbol{\phi}$ using a training set $\{\mathbf{x}_i, \mathbf{y}_i\}$ of input/output pairs. We seek the minimum of the loss function $L[\boldsymbol{\phi}]$:

$$\begin{aligned}\hat{\boldsymbol{\phi}} &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\big[\mathrm{L}[\boldsymbol{\phi}]\big] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[\sum_{i=1}^{I} l_i[\mathbf{x}_i, \mathbf{y}_i]\right], \end{aligned} \tag{9.1}$$

where the individual terms $l_i[\mathbf{x}_i, \mathbf{y}_i]$ measure the mismatch between the network predictions $\mathrm{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ and output targets $\mathbf{y}_i$ for each training pair. To bias this minimization

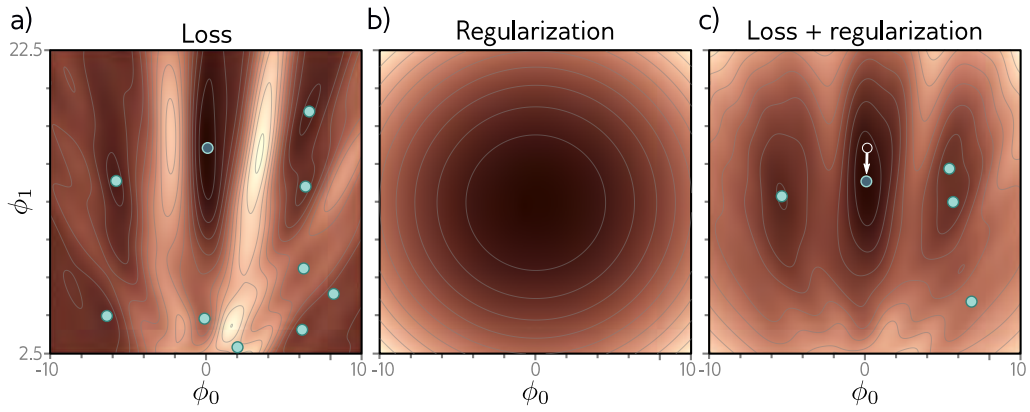

Figure 9.1 Explicit regularization. a) Loss function for Gabor model (see section 6.1.2). Cyan circles represent local minima, green circle represents the global minimum. b) The regularization term favors parameters close to the center of the plot by adding an increasing penalty as we move away from this point. c) The final loss function is the sum of the original loss function plus the regularization term. This surface has fewer local minima, and the global minimum has moved to a different position (arrow shows change).

towards certain solutions, we add an extra term:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmin}} \left[\sum_{i=1}^{I} l_i[\mathbf{x}_i, \mathbf{y}_i] + \lambda \cdot \mathrm{g}[\boldsymbol{\phi}] \right], \tag{9.2}$$

where $\mathrm{g}[\boldsymbol{\phi}]$ is a function that returns a scalar that takes a larger value when the parameters are less preferred. The term λ is a positive scalar that controls the relative contribution of the original loss function and the regularization term. The minima of the regularized loss function usually differ from those in the original, and so the training procedure converges to different parameter values (figure 9.1).

9.1.1 Probabilistic interpretation

Regularization can be viewed from a probabilistic perspective. Section 5.1 shows how loss functions are constructed from the maximum likelihood criterion:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmax}} \left[\prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{x}_i, \boldsymbol{\phi}) \right]. \tag{9.3}$$

The regularization term can be considered as a prior $Pr(\boldsymbol{\phi})$ that represents knowledge about the parameters before we observe the data:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmax}} \left[\prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{x}_i, \boldsymbol{\phi}) Pr(\boldsymbol{\phi}) \right]. \tag{9.4}$$

Moving back to the negative log-likelihood loss function by taking the log and multiplying by minus one, we see that $\lambda \cdot \mathrm{g}[\boldsymbol{\phi}] = -\log[Pr(\boldsymbol{\phi})]$.

9.1.2 L2 regularization

This discussion has sidestepped the question of exactly which solutions the regularization term should penalize (or equivalently that the prior should favor). Since neural networks are used in an extremely broad range of applications, these can only be very generic preferences. The most used commonly regularization term is the *L2 norm*, which penalizes the sum of the squares of the parameter values:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmin}} \left[\mathrm{L}[\boldsymbol{\phi}, \{\mathbf{x}_i, \mathbf{y}_i\}] + \lambda \sum_j \phi_j^2 \right], \tag{9.5}$$

where j indexes the parameters. This is also referred to as *Tikhonov regularization*, *ridge regression*, or *Frobenius norm regularization* when applied to matrices.

Problems 9.1-9.2

For neural networks, L2 regularization is usually applied to the weights but not the biases and is hence referred to as a *weight decay* term. The effect is to encourage smaller weights, so the output function is smoother. To see this, consider that the output prediction is a weighted sum of the activations at the last hidden layer. If the weights have a smaller magnitude, then the output will vary less. The same logic applies to the computation of the pre-activations at the last hidden layer, and so on progressing backward through the network. In the limit, if we forced all the weights to be zero, the network would produce a constant output determined by final bias parameter.

Figure 9.2 shows the effect of fitting the simplified network from figure 8.4 with weight decay and different values of the regularization coefficient λ. When λ is small, it has little effect. However, as λ increases, the fit to the data becomes less accurate, and the function becomes smoother. This might improve the test performance for two reasons:

- If the network is overfitting, then adding the regularization term means that the network must trade off slavish adherence to the data against the desire to be smooth. One way to think about this is that the error due to variance reduces (the model no longer needs to pass through every data point), at the cost of increased bias (the model can only describe smooth functions).
- When the network is over-parameterized, some of the extra model capacity describes areas where there were no training data. Here, the regularization term will favor functions that smoothly interpolate between the nearby points. This is reasonable behavior in the absence of knowledge about the true function.

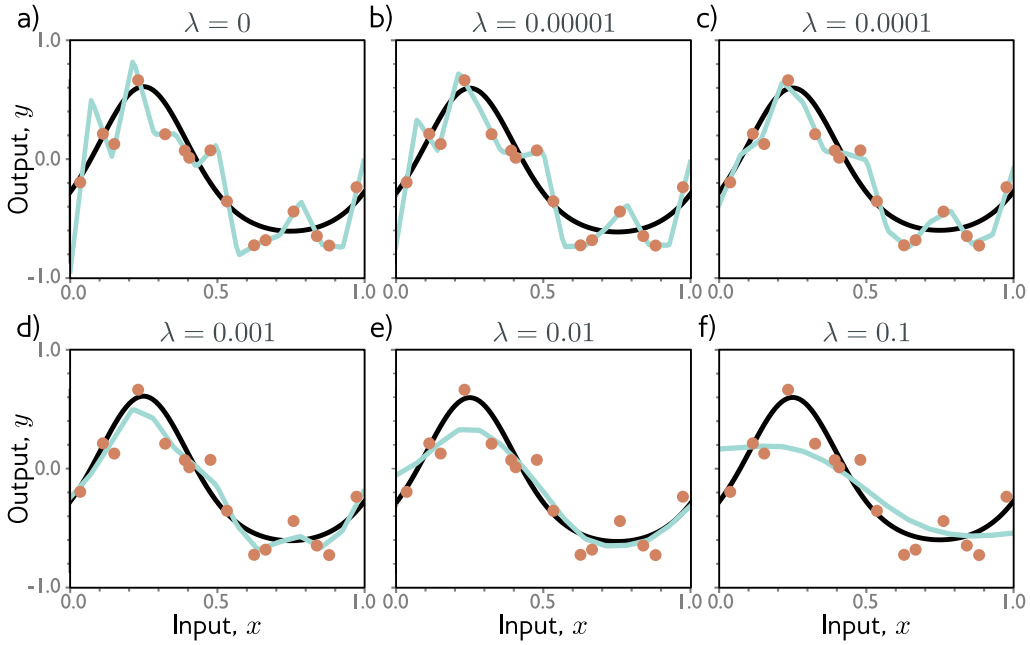

Figure 9.2 L2 regularization in simplified network (figure 8.4). a–f) Fitted functions as we increase the regularization coefficient λ. The black curve is the true function, the orange circles are the noisy training data, and the cyan curve is the fitted model. For small λ (panels a–b), the fitted function passes exactly through the data points. For intermediate λ (panels c–d), the function is smoother and actually more similar to the ground truth. For large λ (panels e–f), the fitted function is smoother than the ground truth, so the fit is worse.

9.2 Implicit regularization

An intriguing recent finding is that neither gradient descent nor stochastic gradient descent descend neutrally to the minimum of the loss function; each exhibits a preference for some solutions over others. This is known as *implicit regularization*.

9.2.1 Implicit regularization in gradient descent

Consider a continuous version of gradient descent where the step size is infinitesimal. The change in parameters $\boldsymbol{\phi}$ will be governed by the differential equation:

$$\frac{\partial \boldsymbol{\phi}}{\partial t} = -\frac{\partial L}{\partial \boldsymbol{\phi}}. \tag{9.6}$$

Gradient descent approximates this process with a series of discrete steps:

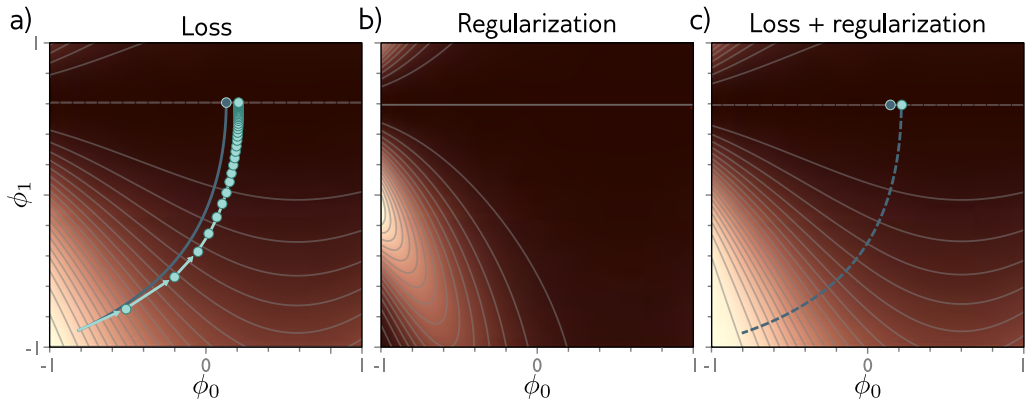

Figure 9.3 Implicit regularization in gradient descent. a) Loss function with family of global minima on horizontal line $\phi_1 = 0.61$. Dashed blue line shows continuous gradient descent path starting in bottom left corner. Cyan trajectory shows discrete gradient descent with step size 0.1 (first few steps shown explicitly as arrows). The finite step size causes the paths to diverge and to reach a different final position. b) This disparity can be approximated by adding a regularization term that penalizes the squared gradient magnitude to the continuous gradient descent loss function. c) After adding this term, the continuous gradient descent path converges to the same place that the discrete one did on the original function.

$$\boldsymbol{\phi}_{t+1} = \boldsymbol{\phi}_t - \alpha \frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}, \tag{9.7}$$

where α is the step size. The discretization means that the algorithm deviates from the continuous path (figure 9.3).

This deviation can be understood by deriving a modified loss term $\tilde{L}$ for the continuous case that arrives at the same place as the discretized version on the original loss L. It can be shown (see end of chapter) that this modified loss is:

$$\tilde{\mathrm{L}}_{GD}[\boldsymbol{\phi}] = \mathrm{L}[\boldsymbol{\phi}] + \frac{\alpha}{4}\left\|\frac{\partial L}{\partial \boldsymbol{\phi}}\right\|^2. \tag{9.8}$$

In other words, the trajectory of the discrete version will be repelled from places where the norm of the gradient is very large, and the surface is steep. This does not affect the position of the minima (where the gradients are zero) but does modify the effective loss function elsewhere and so causes the solution to take a different trajectory and potentially converge to a different minimum. Implicit regularization due to gradient descent is probably responsible for the observation that full batch gradient descent generalizes better with larger step sizes (figure 9.5a).

9.2.2 Implicit regularization in stochastic gradient descent

A similar analysis can be applied to stochastic gradient descent. Now we seek a modified loss function such that the continuous version reaches the same place as the average of the possible random SGD updates. This can be shown to be:

$$\begin{aligned}\tilde{\mathrm{L}}_{SGD}[\boldsymbol{\phi}] &= \tilde{\mathrm{L}}_{GD}[\boldsymbol{\phi}] + \frac{\alpha}{4B}\sum_{b=1}^{B}\left\|\frac{\partial L_b}{\partial \boldsymbol{\phi}} - \frac{\partial L}{\partial \boldsymbol{\phi}}\right\|^2 \\ &= \mathrm{L}[\boldsymbol{\phi}] + \frac{\alpha}{4}\left\|\frac{\partial L}{\partial \boldsymbol{\phi}}\right\|^2 + \frac{\alpha}{4B}\sum_{b=1}^{B}\left\|\frac{\partial L_b}{\partial \boldsymbol{\phi}} - \frac{\partial L}{\partial \boldsymbol{\phi}}\right\|^2. \end{aligned} \tag{9.9}$$

Here, L_b is the loss for the b^{th} of the B batches in an epoch and both L and L_b now represent the means of the I individual losses in the full dataset and the $|\mathcal{B}|$ individual losses in the batch, respectively:

$$L = \frac{1}{I}\sum_{i=1}^{I} l_i[\mathbf{x}_i, y_i] \qquad \text{and} \qquad L_b = \frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}_b} l_i[\mathbf{x}_i, y_i]. \tag{9.10}$$

Equation 9.9 reveals an extra regularization term, which corresponds to the variance of the gradients of the batch losses L_b. In other words, SGD implicitly favors places where the gradients are stable (where all the batches agree on the slope). Once more this modifies the trajectory of the optimization process (figure 9.4) but does not necessarily change the position of the global minimum; if the model is over-parameterized, then it may fit all the training data exactly and so all these gradient terms will all be zero.

SGD generalizes better than gradient descent and smaller batch sizes usually perform better than larger ones (figure 9.5b). One possible explanation is that the inherent randomness allows the algorithm to reach different parts of the loss function. However, it's also possible that some or all of this performance increase is due to implicit regularization; this encourages solutions where all the data fits well (and so the batch variance is small) rather than solutions where some of the data fit extremely well and other data less well (perhaps with the same overall loss, but with larger batch variance). The former solutions are likely to generalize better.

9.3 Heuristics to improve performance

We've seen that adding explicit regularization terms encourages the training algorithm to find a good solution by adding extra terms to the loss function. This also occurs implicitly as an unintended (but seemingly helpful) byproduct of stochastic gradient descent. In this section, we discuss other heuristic methods that have been used to improve generalization.

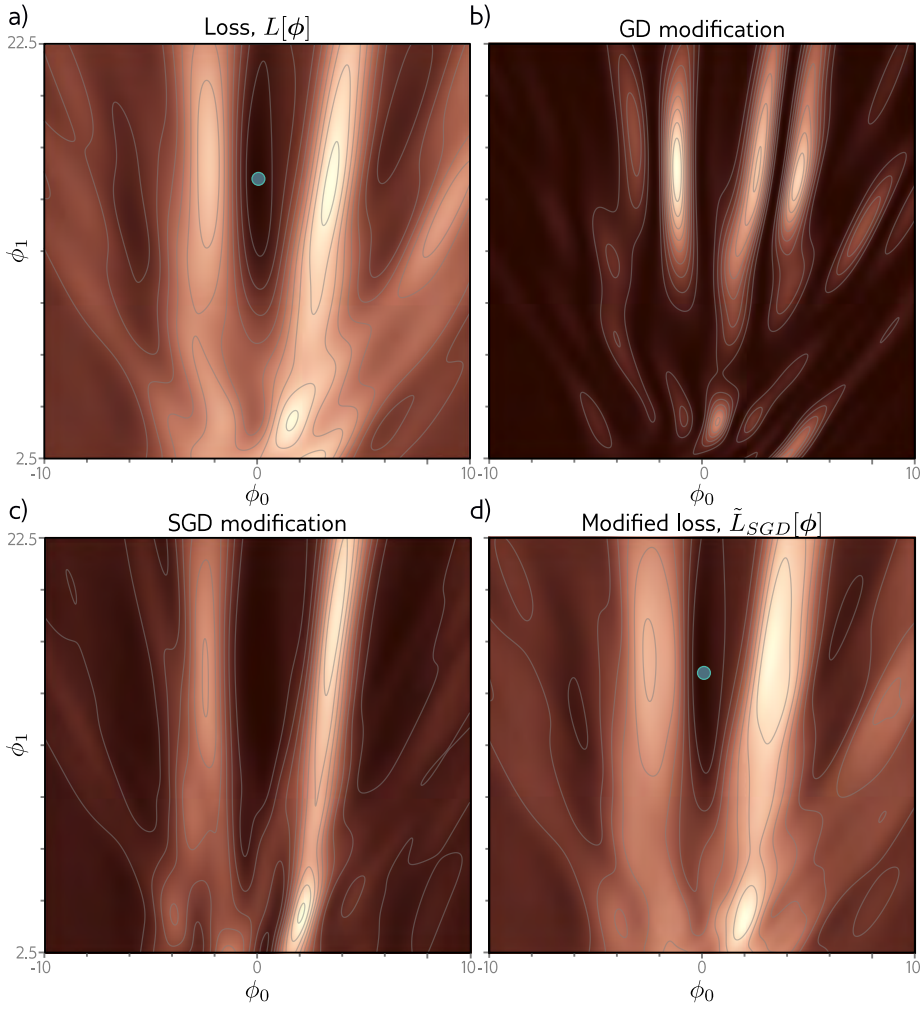

Figure 9.4 Implicit regularization for stochastic gradient descent. a) Original loss function for Gabor model (section 6.1.2). b) Implicit regularization term from gradient descent penalizes the squared gradient magnitude. c) Additional implicit regularization from stochastic gradient descent penalizes the variance of the batch gradients. d) Modified loss function (sum of original loss plus two implicit regularization components).

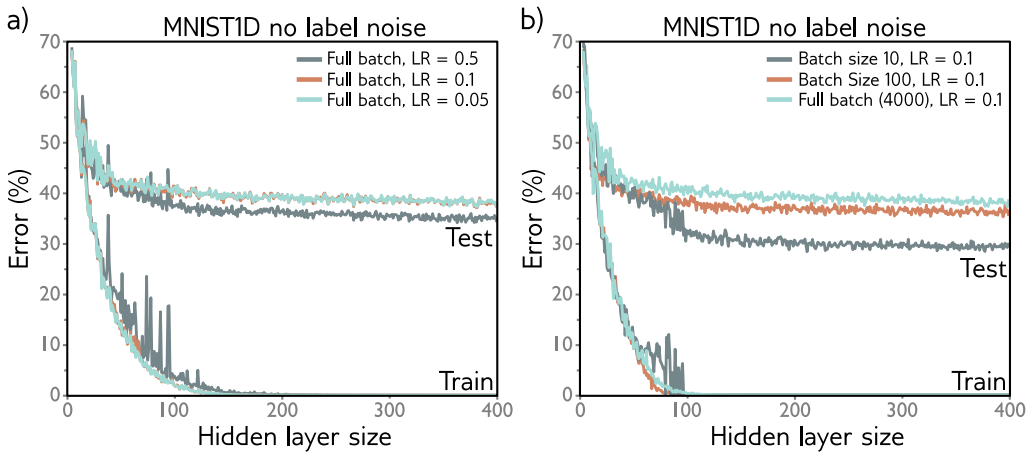

Figure 9.5 Effect of learning rate and batch size for 4000 training examples and 4000 test examples of 1D MNIST data (see figure 8.1) for a neural network with two hidden layers. a) Performance is better for large learning rates than for intermediate or small ones (full batch gradient descent case shown). In each case, the number of iterations is 6000× the learning rate so each solution has the opportunity to move the same distance. b) Performance is superior for smaller batch sizes. In each case the number of iterations was chosen so that the training data were memorized at roughly the same model capacity.

9.3.1 Early stopping

As the name suggests, *early stopping* refers to the practice of stopping the training procedure before it has fully converged. This can reduce overfitting because the model may have already captured the coarse shape of the underlying function, but not yet had time to overfit to the noise (figure 9.6).

One way of thinking about this is that since the weights are initialized to small values (see section 7.3), they simply don't have time to become large, and so early stopping has a similar effect to explicit L2 regularization. A different view is that early stopping reduces the effective model complexity. Hence, we move back down the bias/variance trade-off curve from the critical region and performance improves (see figures 8.9 and 8.10).

Early stopping has a single hyperparameter, which is the number of steps after which learning is terminated. As usual, this is chosen empirically using a validation set (section 8.5). However, for early stopping, the hyperparameter can be chosen without the need to train multiple models. The model is trained once, the performance on the validation set is monitored every T iterations, and the associated models are stored. The stored model where the validation performance was best is selected.

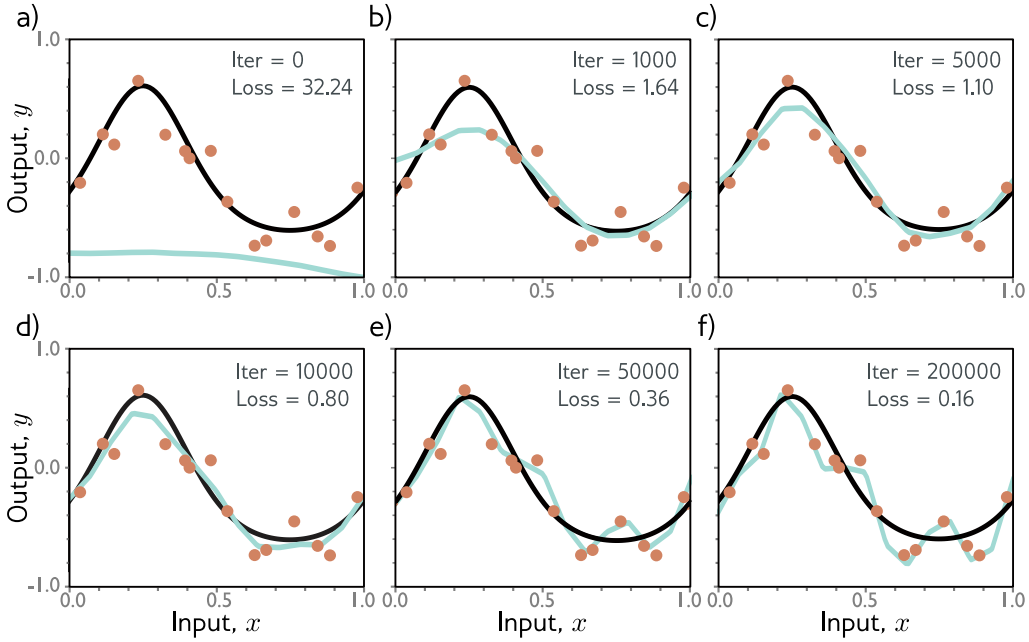

Figure 9.6 Early stopping. a) Simplified shallow network model with 14 linear regions (figure 8.4) is initialized randomly (cyan curve) and trained with SGD using a batch size of five and a learning rate of 0.05. b-d) As training proceeds, the function first captures the coarse structure of the true function (black curve), before e-f) overfitting to the noisy training data (orange points). Although the training loss continues to decrease throughout this process, the learned models in panels (c) and (d) are closest to the true underlying function and will generalize better on average to test data than those in panels (e) or (f).

9.3.2 Ensembling

A completely different approach to reducing the generalization gap between training and test data, is to build several models and average their predictions. A group of such models is known as an *ensemble*. This technique reliably improves test performance at the cost of training and storing multiple models and performing inference multiple times.

The models can be combined by taking the mean of the outputs (for regression problems) or the mean of the pre-softmax activations (for classification problems). The underlying assumption here is that the errors in the different models are independent and will cancel out. Alternatively, we can take the median of the outputs (for regression problems) or the most frequent predicted class (for classification problems) to make the predictions more robust.

One way to train different models is just to use different random initializations. This may particularly help in regions of the input space that are far from the training data and

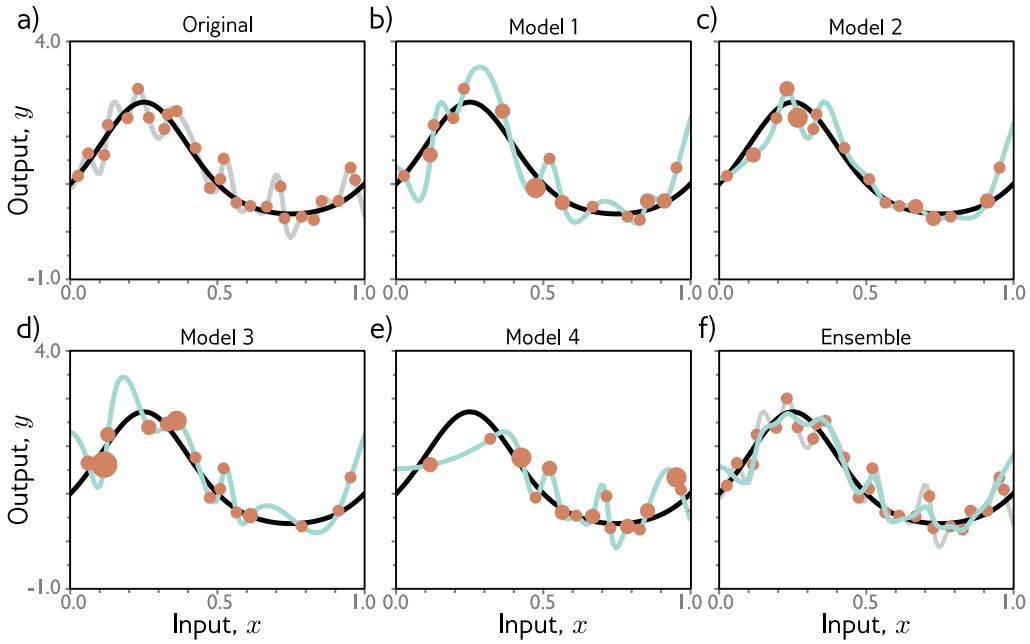

Figure 9.7 Ensemble methods. a) Fitting a single model (gray curve) to the entire dataset (orange points). b-e) An ensemble of four models is created by re-sampling the data with replacement (bagging) four times and fitting a model to each (size of orange point indicates number of times the data point was re-sampled). f) When we average these predictions (cyan curve), the result is smoother than the result from panel (a) for the full dataset (gray curve) and will probably generalize better.

where the fitted function is relatively unconstrained. Different models may produce very different predictions in these regions and so the average of several models may generalize better than any individual model.

A second approach is to generate several different datasets by re-sampling the training data with replacement and training a different model from each. This is known as *bootstrap aggregating* or *bagging* for short (figure 9.7). It has the effect of smoothing out the data; if a data point is not present in one training set, then the model will interpolate from nearby points; hence, if that point was an outlier, then the fitted function will be generally more moderate in this region. Other approaches include training models with different hyperparameters, or training completely different families of models.

9.3.3 Dropout

Dropout randomly clamps a subset (typically 50%) of hidden units to zero at each iter-

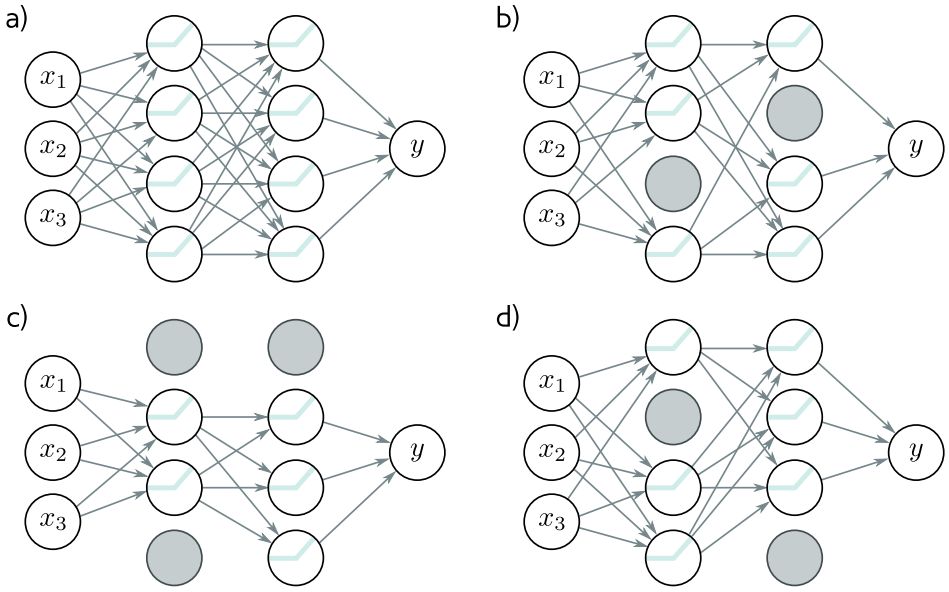

Figure 9.8 Dropout. a) Original network. b-d) At each training iteration, a random subset of hidden units is clamped to zero (gray nodes). The result is that the incoming and outgoing weights from these units have no effect and so we are training with a slightly different network each time.

ation of SGD (figure 9.8). This makes the network less dependent on any given hidden unit; this encourages the weights to have smaller magnitudes so that the change in the function due to the presence or absence of the hidden unit is reduced.

This technique has the positive benefit that it can eliminate kinks in the function that are far from the training data and so don't affect the loss. For example, consider three hidden units that become active sequentially as we move along the curve (figure 9.9a). The first hidden unit causes a large increase in the slope. A second hidden unit decreases the slope so that the function goes back down. Finally, the third unit cancels out this decrease and returns the subsequent curve to its original trajectory. These three units conspire to make an undesirable local change in the function. This will not change the training loss but is unlikely to generalize well.

When several units conspire to make an undesirable local change, eliminating one of them (as would happen in dropout) causes a huge change in the output function that is propagated to the half-space where that unit was active (figure 9.9b). A subsequent gradient descent step will attempt to compensate for the change that this induces, and over time such dependencies will be eliminated. The overall effect is that large unnecessary changes between training data points are gradually removed even though they contribute nothing to the loss (figure 9.9).

At test time, we can run the network as normal with all the hidden units active;

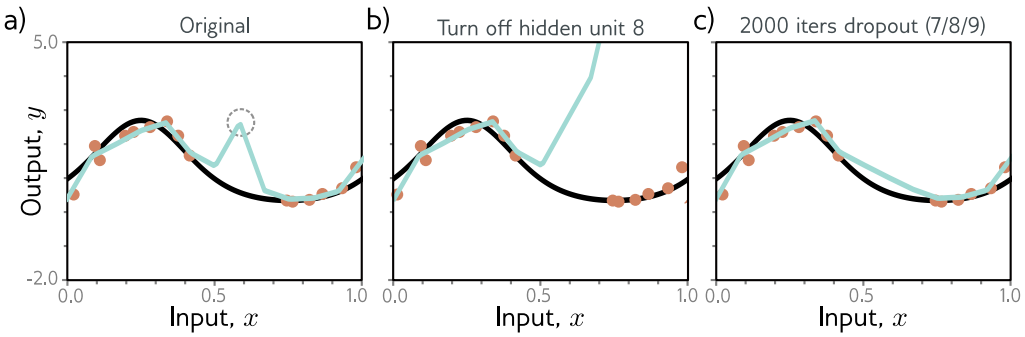

Figure 9.9 Dropout mechanism. a) There is an undesirable kink in the curve caused by a sequential increase in the slope, decrease in the slope (at circled joint), and then another increase to return the curve to its original trajectory. Here we are using full-batch gradient descent and the model already fits the data as well as possible, so further training won't remove the kink. b) Consider what happens if we remove the hidden unit that produced the circled joint in panel (a) as might happen using dropout. Without the decrease in the slope, the right-hand side of the function takes an upwards trajectory and a subsequent gradient descent step will aim to compensate for this change. c) Curve after 2000 iterations of (i) randomly removing one of the three hidden units that cause the kink, and (ii) performing a gradient descent step. The kink does not affect the loss but is nonetheless removed by this approximation of the dropout mechanism.

however, the network now has more hidden units than it was trained with at any given iteration, so we multiply the weights by the dropout probability to compensate. This is known as the *weight scaling inference rule*. A different approach to inference is to use *Monte Carlo dropout*, in which we run the network multiple times with different random subsets of units clamped to zero (as in training) and combine the results. This is closely related to ensembling in that every random version of the network is a different model; however, here we do not have to train or store multiple networks.

9.3.4 Applying noise

Dropout can be interpreted as applying multiplicative Bernoulli noise to the network activations. This leads to the idea of applying noise to other parts of the network during training, with the goal of making the final model more robust.

Problem 9.3

One option is to add noise to the input data; this has smooths out the learned function (figure 9.10). For regression problems, it can be shown to be equivalent to adding a regularizing term that penalizes the derivatives of the output of the network with respect to its input. An extreme variant is *adversarial training*, in which the optimization algorithm actively searches for small perturbations of the input that cause large changes to the output. These can be thought of as worst-case additive noise vectors.

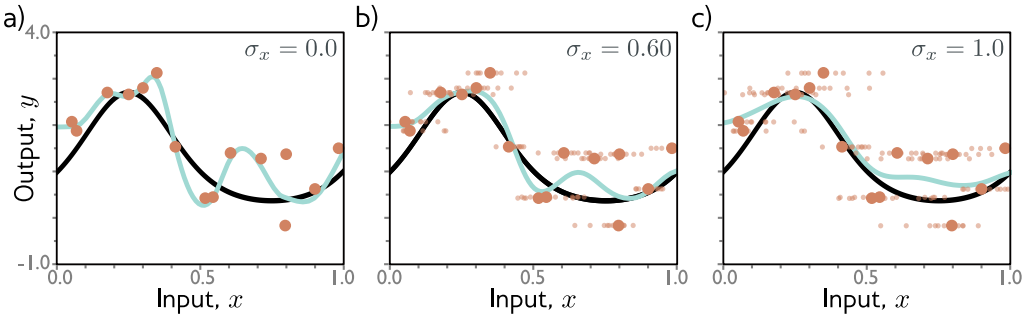

Figure 9.10 Adding noise to inputs. At each step of SGD, random noise with variance σ_x^2 is added to the batch data. a-c) Fitted model with different levels of noise (small dots represent ten samples). Adding more noise smooths out the fitted function (cyan line).

A second possibility is to add noise to the weights. This encourages the network to make sensible predictions even for small perturbations of the weights. The result is that the training converges to local minima that are in the middle of wide, flat regions, where changing the individual weights does not matter much.

Finally, we can perturb the labels. The maximum-likelihoodcriterion for multi-class classification aims to predict the correct class with absolute certainty (equation 5.27). To do this, the final network activations (i.e., before the softmax function) must be pushed to very large values for the correct class and very small values for the wrong classes.

We could discourage this overconfident behavior by assuming that a proportion ρ of the training labels are incorrect and belong with equal probability to the other classes. In principle, this could be done by randomly changing the labels at each iteration of training. However, the same end can be achieved by changing the loss function to minimize the cross entropy between the predicted distribution and a distribution where the true label has probability $1-\rho$ and the other classes have equal probability. This is known as *label smoothing* and improves generalization in diverse scenarios.

Problem 9.4

9.3.5 Bayesian approaches

The maximum likelihood approach is generally overconfident; in the training phase, it selects the most likely parameters and bases its predictions on the model defined by these. However, it may be that many parameter values are broadly compatible with the data and are only slightly less likely. The Bayesian approach treats the parameters as unknown variables and computes a distribution $Pr(\boldsymbol{\phi}|\{\mathbf{x}_i, \mathbf{y}_i\})$ over these parameters $\boldsymbol{\phi}$ conditioned on the training data $\{\mathbf{x}_i, \mathbf{y}_i\}$ using Bayes' rule:

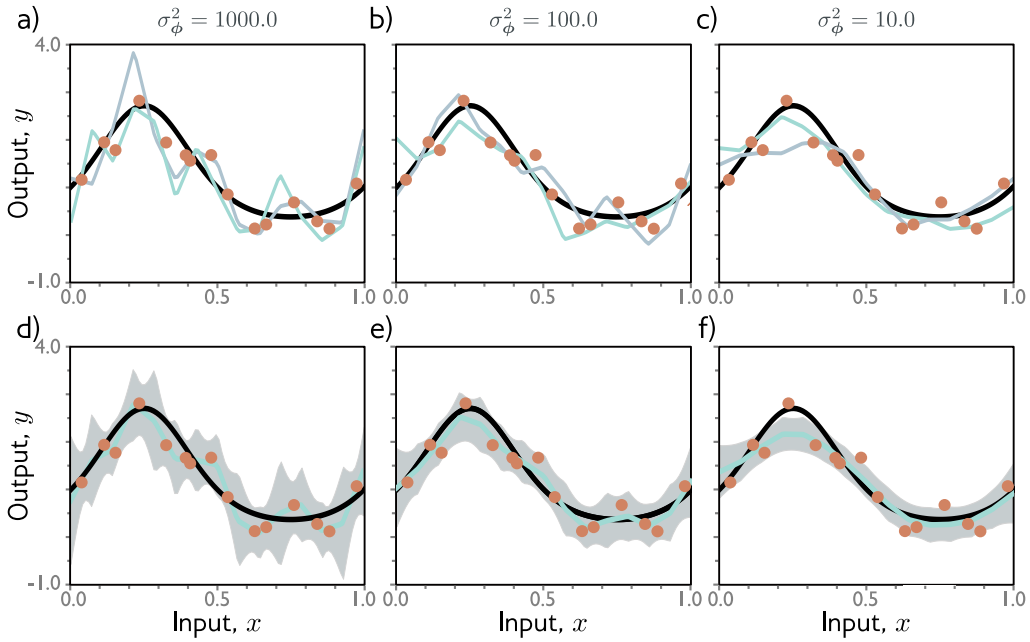

Figure 9.11 Bayesian approach for simplified network model (figure 8.4). Rather than finding one set of parameters $\boldsymbol{\phi}$, the Bayesian approach treats the parameters as uncertain. The posterior probability $Pr(\boldsymbol{\phi}|\{\mathbf{x}_i, \mathbf{y}_i\})$ for a set of parameters is determined by their compatibility with the data $\{\mathbf{x}_i, \mathbf{y}_i\}$ and a prior distribution $Pr(\boldsymbol{\phi})$. a–c) Two possible sets of parameters (cyan curves) sampled from the posterior distribution using normally distributed priors with mean zero and three different variances. When the prior variance is small, the parameters also tend to be small, and the functions become smoother. d–f) Inference proceeds by taking a weighted sum over all possible parameter values where the weights are the posterior probabilities. Accordingly, we get both a mean prediction (cyan curves) and associated uncertainty (gray region is two standard deviations).

$$Pr(\boldsymbol{\phi}|\{\mathbf{x}_i, \mathbf{y}_i\}) = \frac{\prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{x}_i, \boldsymbol{\phi}) Pr(\boldsymbol{\phi})}{\int \prod_{i=1}^{I} Pr(\mathbf{y}_i|\mathbf{x}_i, \boldsymbol{\phi}) Pr(\boldsymbol{\phi}) d\boldsymbol{\phi}}, \tag{9.11}$$

where $Pr(\boldsymbol{\phi})$ is the prior probability of the parameters, and the denominator is a normalizing term. Accordingly, every possible set of parameters in the model family is assigned a probability (figure 9.11).

The prediction $\mathbf{y}$ for new input $\mathbf{x}$ is an infinite weighted sum (i.e., an integral) of the predictions for each parameter set, where the weights are the associated probabilities:

$$Pr(\mathbf{y}|\mathbf{x}, \{\mathbf{x}_i, \mathbf{y}_i\}) = \int Pr(\mathbf{y}|\mathbf{x}, \boldsymbol{\phi}) Pr(\boldsymbol{\phi}|\{\mathbf{x}_i, \mathbf{y}_i\}) d\boldsymbol{\phi}. \tag{9.12}$$

This is effectively an infinite weighted ensemble, where the weight depends on (i) their agreement with the data and (ii) the prior probability of the parameters.

The Bayesian approach is elegant and can provide more robust predictions than those that derive from maximum likelihood. Unfortunately, for complex models like neural networks, there is no practical way to represent the full probability distribution over the parameters or to integrate over it during the inference phase. Consequently, all current methods of this type make approximations of some kind, and typically these add considerable complexity to learning and inference.

9.3.6 Transfer learning and multi-task learning

When training data for the primary task are limited, other datasets can be exploited to improve performance. In *transfer learning* (figure 9.12a), the network is trained to perform a related secondary task for which data are more plentiful. The resulting model is subsequently adapted to the primary task. This is typically done by removing the last layer and adding one or more layers that produce a suitable output. The first part of the model may be fixed and the new layers trained for the primary task, or we may *fine-tune* the entire model.

The core idea is that network will build a good internal representation of the data from the secondary task, and this can subsequently be exploited for the primary task. Equivalently, transfer learning can be viewed as initializing most of the parameters of the final network in a sensible part of the space that is likely to produce a good solution.

A related technique is *multi-task* learning (figure 9.12b); here the network is trained to solve several problems concurrently. For example, the network might take an image and simultaneously learn to segment the scene, estimate the pixel-wise depth, and predict a caption describing the image. All of these tasks require some understanding of the image and when learned simultaneously, the model performance for each may improve.

9.3.7 Self-supervised learning

The above discussion assumes that we have plentiful data for a secondary task or data for multiple tasks to be learned concurrently. If not, then we can create large amounts of "free" labeled data using *self-supervised* learning and use this for transfer learning. There are two families of methods for self-supervised learning: *generative* and *contrastive*.

In generative self-supervised learning, part of each data example is masked and the secondary task is to predict the missing part (figure 9.12c). For example, we might use a corpus of unlabeled images and a secondary task that aims to *inpaint* (fill in) missing parts of the image (figure 9.12c). Similarly, we might use a large corpus of text and remove one or two words from each sentence. We train the network to predict the missing words and then fine-tune it for the real language task that we are interested in (see chapter 12).

In contrastive self-supervised learning, two versions of each unlabeled example are presented, where one has been distorted in some way. The system is trained to predict

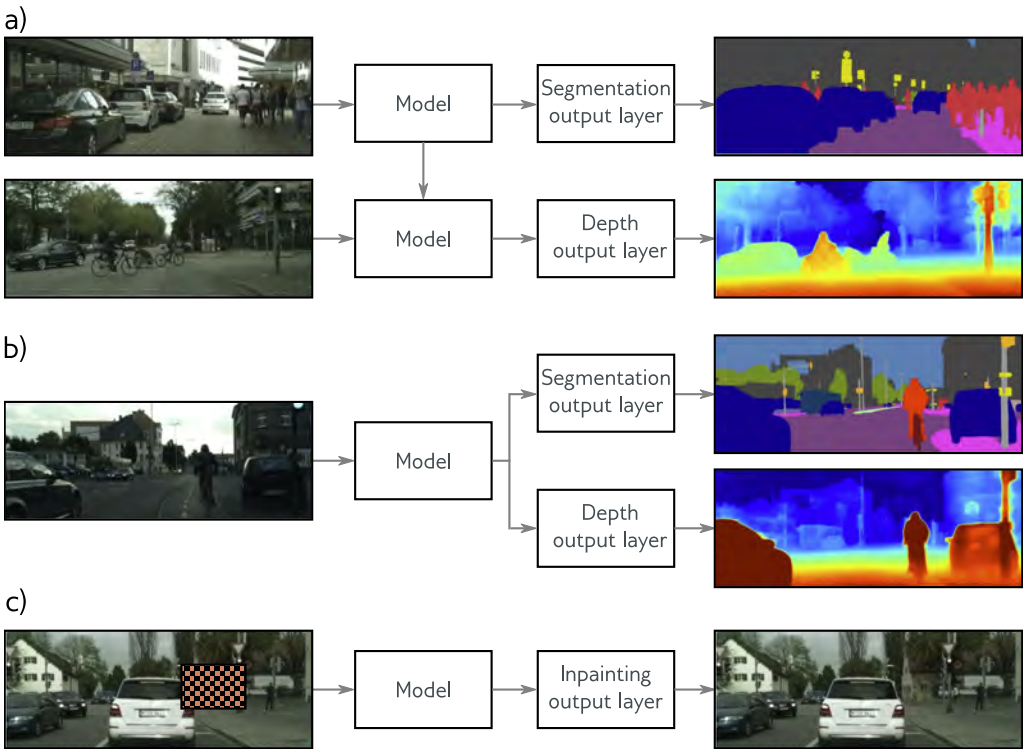

Figure 9.12 Transfer, multi-task, and self-supervised learning. a) Transfer learning is used when we have limited labeled data for the primary task (here depth estimation), but plentiful data for a secondary task (here segmentation). We train a model for the secondary task, remove the final layers, and replace them with new layers appropriate to the primary task. We then train just the new layers or fine-tune the entire network for the primary task. The network learns a good internal representation from the secondary task that is then exploited for the primary task. b) In multi-task learning, we train a model to perform multiple tasks simultaneously, hoping that performance on each will improve. c) In generative self-supervised learning, we remove part of the data and train the network to complete the missing information. This permits transfer learning when no labels are available. Images from Cordts et al. (2016).

Figure 9.13 Data augmentation. For certain types of problem, it is easy to transform each data example multiple times to augment the dataset. a) Original image. b-h) Various geometric and photometric transformations of this image. For an image classification problem, all these modified versions still have the same label 'bird'. Adapted from Wu et al. (2015a).

which is the original. For example, we might use a corpus of unlabeled images where the secondary task is to identify which version of the image is upside-down. Similarly, we might use a large corpus of text, where the secondary task is to determine whether two sentences followed one another or not in the original text.

9.3.8 Augmentation

Transfer learning improves performance by exploiting a different dataset. Multi-task learning improves performance by exploiting additional labels. A third option is to expand the original dataset. For many tasks, we can transform each input data example in such a way that the label stays the same. For example, an image classification task might aim to determine if there is a bird in an image (figure 9.13). Here, we could rotate, flip, blur, or manipulate the color balance of the image and the label "bird" remains valid. Similarly, for tasks where the input is text, we can substitute synonyms or translate to another language and back again. For tasks where the input is audio, we can amplify or attenuate different frequency bands.

Generating extra training examples using this approach is known as *data augmenta-*

tion. The aim is to teach the machine learning system to be indifferent to these irrelevant transformations of the data.

9.4 Summary

Explicit regularization involves adding an extra term to the loss function that changes the position of the minimum. The term can be interpreted a prior probability over the parameters. Stochastic gradient descent with a finite step size does not neutrally descend to the minimum of the loss function. This bias can be interpreted as adding additional terms to the loss function, and this is known as implicit regularization.

There are also many heuristics that are intended to improve the generalization performance of deep learning methods, including early stopping, dropout, ensembling, the Bayesian approach, adding noise, transfer learning, multi-task learning, and data augmentation. Considered together, there are four main principles behind these methods (figure 9.14). We can (i) encourage the function to be smoother (e.g., L2 regularization), (ii) increase the effective amount of data (e.g., data augmentation), (iii) combine multiple models (e.g., ensembling), or (iv) search for wider minima in the loss function (e.g., applying noise to network weights).

One further way to improve model performance is to share parameters between different parts of the model. For example, in an image segmentation task, it is inefficient to separately learn which pattern of pixels constitutes a tree at every location in the image. This is the topic of chapter 10.

Notes

An overview and taxonomy of regularization techniques in deep learning can be found in Kukačka et al. (2017). Notably missing from the discussion in this chapter is BatchNorm (Szegedy et al. 2016), which is described in chapter 11.

Regularization: L2 regularization penalizes the sum of squares of the network weights. This encourages the output function to change slowly (i.e., become smoother) and is the most used regularization term. It is sometimes referred to as Frobenius norm regularization as it penalizes the Frobenius norm of the weight matrices. It is often also mistakenly referred to as "weight decay" although this is a separate technique devised by Hanson & Pratt (1988) in which the parameters $\boldsymbol{\phi}$ are updated as:

$$\boldsymbol{\phi} \longleftarrow (1-\lambda')\boldsymbol{\phi} - \alpha\frac{\partial L}{\partial \boldsymbol{\phi}}, \tag{9.13}$$

where as usual α is the learning rate and L is the loss. This is identical to gradient descent, except that the weights are reduced by a factor of $1-\lambda'$ before the gradient update. For standard SGD, weight decay is equivalent to L2 regularization (equation 9.5) with coefficient $\lambda = \lambda'/2\alpha$. However, for Adam the learning rate α is different for each parameter and so L2 regularization

Problem 9.5

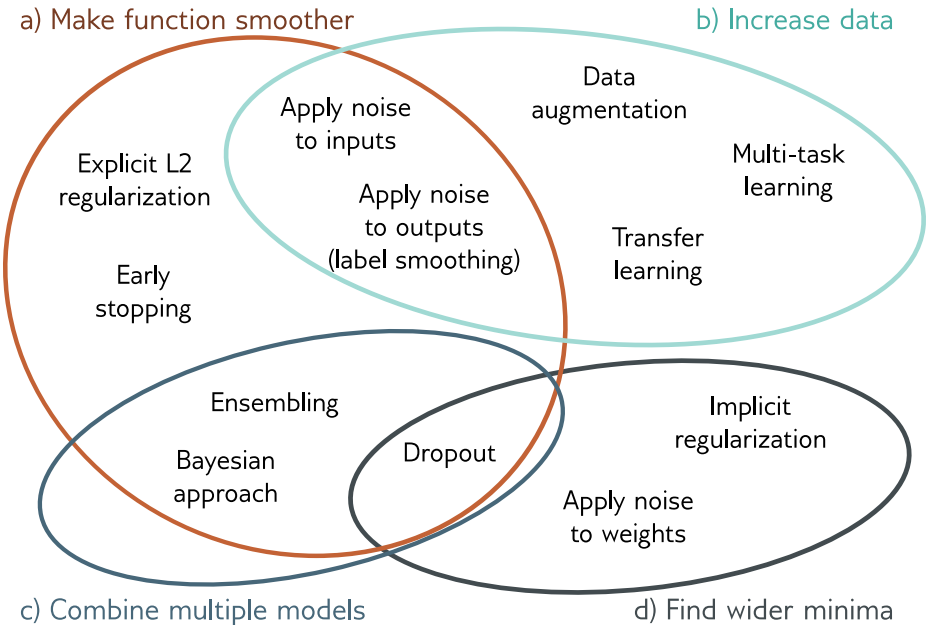

Figure 9.14 Regularization methods. The regularization methods discussed in this chapter aim to improve generalization by one of four mechanisms. a) Some methods aim to make the modeled function smoother. b) Other methods increase the effective amount of data. c) The third group of methods combine multiple models and hence mitigate against uncertainty in the fitting process. d) Finally, the fourth group of methods encourages the training process to converge to a wide minimum where small errors in the estimated parameters are less important.

and weight decay differ. Loshchilov & Hutter (2017) present AdamW which modifies Adam to implement weight decay correctly and show that this improves performance.

A different approach is to encourage sparsity in the weights. The L0 regularization term applies a fixed penalty for every non-zero weight. The overall effect is to "prune" the network. Another approach is to add an L0 regularization term that encourages group sparsity; this might apply a fixed penalty if any of the weights contributing to a given are non-zero. If they are all zero, we can remove this hidden unit, decreasing the model size, and making inference faster.

Appendix C.1.5 Vector norms

Unfortunately, L0 regularization is challenging to implement since the derivative of the regularization term is not smooth, and more sophisticated fitting methods are required (see Louizos et al. 2018). Somewhere between L2 and L0 regularization is L1 regularization or *LASSO* (least absolute shrinkage and selection operator), which imposes a penalty on the absolute values of the weights. L2 regularization somewhat discourages sparsity in that the derivative of the squared penalty decreases as the weight becomes smaller, lowering the pressure to make it smaller still. L1 regularization does not have this disadvantage as the derivative of the penalty is constant. This can produce sparser solutions than L2 regularization but is much easier to optimize than L0 regularization. Sometimes both L1 and L2 regularization terms are both used, and this is termed an *elastic net* penalty.

Problem 9.6

An alternative approach to regularization is to directly modify the gradients of the learning algorithm without ever explicitly formulating a modified loss function (as we already saw in equation 9.13). This approach has been used to promote sparsity during back-propagation (Schwarz et al. 2021).

The evidence on the effectiveness of explicit regularization is mixed. Zhang et al. (2016a) showed that L2 regularization contributes little to generalization. It has been shown that the Lipschitz constant of the network (how fast the function can change as we modify the input) bounds the generalization error (Bartlett et al. 2017a; Neyshabur et al. 2017b). However, the Lipschitz constant depends on the product of the spectral norm of the weight matrices $\boldsymbol{\Omega}_k$, which are only indirectly dependent on the magnitudes of the individual weights. Bartlett et al. (2017a), Neyshabur et al. (2017b), and Yoshida & Miyato (2017) all add terms that indirectly encourage the spectral norms to be smaller. Gouk et al. (2018) take a different approach and develop an algorithm that constrains the Lipschitz constant of the network to be below a certain value.

Appendix C.4 Lipschitz constant

Appendix C.1.6 spectral norm

Implicit regularization in gradient descent: The gradient descent step is

$$\boldsymbol{\phi}_1 = \boldsymbol{\phi}_0 + \alpha \mathbf{g}[\boldsymbol{\phi}_0], \tag{9.14}$$

where $\mathbf{g}[\boldsymbol{\phi}_0]$ is the negative of the gradient of the loss function and α is the step size. As $\alpha \to 0$, the gradient descent process can be described by a differential equation:

$$\frac{\partial \boldsymbol{\phi}}{\partial t} = \mathbf{g}[\boldsymbol{\phi}]. \tag{9.15}$$

For typical step sizes α, the discrete and continuous versions converge to different solutions. We can use *backward error analysis* to find a correction $\mathbf{g}_1[\boldsymbol{\phi}]$ to the continuous version:

$$\frac{\partial \boldsymbol{\phi}}{\partial t} \approx \mathbf{g}[\boldsymbol{\phi}] + \alpha \mathbf{g}_1[\boldsymbol{\phi}] + \ldots, \tag{9.16}$$

so that it gives the same result as the discrete version.

Consider the first two terms of a Taylor expansion of the modified continuous solution $\boldsymbol{\phi}$ around initial position $\boldsymbol{\phi}_0$:

$$\begin{aligned}
\boldsymbol{\phi}[\alpha] &\approx \boldsymbol{\phi} + \alpha \frac{\partial \boldsymbol{\phi}}{\partial t} + \frac{\alpha^2}{2} \frac{\partial^2 \boldsymbol{\phi}}{\partial t^2}\bigg|_{\boldsymbol{\phi}=\boldsymbol{\phi}_0} \\
&\approx \boldsymbol{\phi} + \alpha \left(\mathbf{g}[\boldsymbol{\phi}] + \alpha \mathbf{g}_1[\boldsymbol{\phi}]\right) + \frac{\alpha^2}{2} \left(\frac{\partial \mathbf{g}[\boldsymbol{\phi}]}{\partial \boldsymbol{\phi}} \frac{\partial \boldsymbol{\phi}}{\partial t} + \alpha \frac{\partial \mathbf{g}_1[\boldsymbol{\phi}]}{\partial \boldsymbol{\phi}} \frac{\partial \boldsymbol{\phi}}{\partial t} \right)\bigg|_{\boldsymbol{\phi}=\boldsymbol{\phi}_0} \\
&= \boldsymbol{\phi} + \alpha \left(\mathbf{g}[\boldsymbol{\phi}] + \alpha \mathbf{g}_1[\boldsymbol{\phi}]\right) + \frac{\alpha^2}{2} \left(\frac{\partial \mathbf{g}[\boldsymbol{\phi}]}{\partial \boldsymbol{\phi}} \mathbf{g}[\boldsymbol{\phi}] + \alpha \frac{\partial \mathbf{g}_1[\boldsymbol{\phi}]}{\partial \boldsymbol{\phi}} \mathbf{g}[\boldsymbol{\phi}] \right)\bigg|_{\boldsymbol{\phi}=\boldsymbol{\phi}_0} \\
&\approx \boldsymbol{\phi} + \alpha \mathbf{g}[\boldsymbol{\phi}] + \alpha^2 \left(\mathbf{g}_1[\boldsymbol{\phi}] + \frac{1}{2} \frac{\partial \mathbf{g}[\boldsymbol{\phi}]}{\partial \boldsymbol{\phi}} \mathbf{g}[\boldsymbol{\phi}] \right)\bigg|_{\boldsymbol{\phi}=\boldsymbol{\phi}_0},
\end{aligned} \tag{9.17}$$

where in the second line, we have introduced the correction term (equation 9.16), and in the final line, we have removed terms of greater order than α^2.

Note that the first two terms on the right-hand side $\boldsymbol{\phi}_0 + \alpha \mathbf{g}[\boldsymbol{\phi}_0]$ are the same as the discrete update (equation 9.14). Hence, to make the continuous and discrete versions arrive at the same place, the second term on the right-hand side must equal zero allowing us to solve for $\mathbf{g}_1[\boldsymbol{\phi}]$:

$$\mathbf{g}_1[\boldsymbol{\phi}] = -\frac{1}{2}\frac{\partial \mathbf{g}[\boldsymbol{\phi}]}{\partial \boldsymbol{\phi}}\mathbf{g}[\boldsymbol{\phi}]. \tag{9.18}$$

When we optimize neural networks, the evolution function $\mathbf{g}[\boldsymbol{\phi}]$ is the negative of the gradient of the loss so we have:

$$\begin{aligned}\frac{\partial \boldsymbol{\phi}}{\partial t} &\approx \mathbf{g}[\boldsymbol{\phi}] + \alpha \mathbf{g}_1[\boldsymbol{\phi}] \\ &= -\frac{\partial L}{\partial \boldsymbol{\phi}} - \frac{\alpha}{2}\left(\frac{\partial^2 L}{\partial \boldsymbol{\phi}^2}\right)\frac{\partial L}{\partial \boldsymbol{\phi}}.\end{aligned} \tag{9.19}$$

This is the equivalent to performing continuous gradient descent on the loss function:

$$\mathrm{L}_{GD}[\boldsymbol{\phi}] = \mathrm{L}[\boldsymbol{\phi}] + \frac{\alpha}{4}\left\|\frac{\partial L}{\partial \boldsymbol{\phi}}\right\|^2, \tag{9.20}$$

because we retrieve equation 9.19 by taking the derivative of equation 9.20.

This formulation of implicit regularization was developed by Barrett & Dherin (2021) and extended to stochastic gradient descent by Smith et al. (2021). Smith et al. (2020) and others have shown that stochastic gradient descent with small or moderate batch sizes outperforms full batch gradient descent on the test set, and this may in part be due to implicit regularization.

Early stopping: Bishop (1995) and Sjöberg & Ljung (1995) argued that early stopping limits the effective solution space that the training procedure can explore; given that the weights are initialized to small values, this leads to the idea that early stopping helps prevent the weights from getting too large. Goodfellow et al. (2016) show that under a quadratic approximation of the loss function with parameters initialized to zero, early stopping is equivalent to L2 regularization in gradient descent. The effective regularization weight λ can be shown to be approximately $1/(\tau\alpha)$ where α is the learning rate and τ is the early stopping time.

Ensembling: Ensembles can be trained by using different random seeds (Lakshminarayanan et al. 2017), hyperparameters (Wenzel et al. 2020b), or even entirely different families of model. The models can be combined by averaging their predictions, weighting the predictions, or *stacking* (Wolpert 1992), in which the results are combined using another machine learning model. Lakshminarayanan et al. (2017) showed that averaging the output of independently trained networks can improve accuracy, calibration, and robustness. Conversely, Frankle et al. (2020) showed that if we average together the weights to make one model, then the network fails. Fort et al. (2020) compared ensembling solutions that resulted from different initializations with ensembling solutions that were generated from the same original mode. For example, in the latter case, they consider exploring around the solution in a limited subspace to find other good nearby points. They found that both techniques provide complementary benefits, but that genuine ensembling from different random starting points provides a bigger benefit.

Appendix C.1.3 subspaces

An efficient way of ensembling is to combine models from the intermediate stages of training. To this end, Izmailov et al. (2018) introduce *stochastic weight averaging*, in which the model weights are sampled at different time steps and averaged together. As the name suggests, *snapshot ensembles* (Huang et al. 2017a) also store the models from different time steps and average their predictions. To increase the diversity of these models, they cyclically increase and decrease the learning rate. Garipov et al. (2018) observed that different minima of the loss function are often connected by a low-energy path. Motivated by this observation they developed a method that explores low-energy regions around an initial solution to provide diverse models without having to completely retrain. This method is known as *fast geometric ensembling*. A review of ensembling methods for deep learning can be found in Ganaie et al. (2021).

Dropout: Dropout was first introduced by Hinton et al. (2012b) and Srivastava et al. (2014). Dropout is applied at the level of hidden units. Dropping a hidden unit has the same effect as temporarily setting all the incoming and outgoing weights and the bias to zero. Wan et al. (2013) generalized dropout by randomly setting individual weights to zero.

Gal & Ghahramani (2015) and Kendall & Gal (2017) proposed Monte-Carlo dropout, in which inference is computed with several dropout patterns and the results are averaged together. Gal & Ghahramani (2015) argued that this can be interpreted as Bayesian inference.

Dropout is equivalent to applying multiplicative Bernoulli noise to the hidden units. Similar benefits have been shown to be derived from using other noise distributions including the normal distribution (Srivastava et al. 2014; Shen et al. 2017), uniform distribution (Shen et al. 2017), and beta distribution (Liu et al. 2019).

Adding noise: Bishop (1995) and An (1996) both added Gaussian noise to the network inputs to improve performance. Bishop (1995) showed that this is equivalent to weight decay. An (1996) also investigated adding noise to the weights. DeVries & Taylor (2017a) added Gaussian noise to the hidden units. Xu et al. (2015) apply noise in a different way with the *randomized ReLU* by making the activation functions stochastic.

Finding wider minima: It is thought that wider minima in the loss function generalize better. Here, the exact values of the weights are less important and so performance should be robust to errors in their estimates. One of the reasons that applying noise to parts of the network during training is effective is that it encourages the network to be indifferent to their exact values.

Chaudhari et al. (2017) develop a variant of SGD that deliberately biases the optimization towards flat minima, which they call *entropy SGD*. The basic idea is to incorporate the local entropy as a term in the loss function. In practice, this takes the form of one SGD like update within another. Keskar et al. (2017) showed that SGD finds wider minima as the batch size is reduced. This may be because of the batch variance term that results from implicit regularization by SGD.

Appendix B.1.5 entropy

Ishida et al. (2020) use a technique named *flooding*, in which they intentionally prevent the training loss from becoming zero. This encourages the solution to perform a random walk over the loss landscape and drift into a flatter area with better generalization.

Label smoothing: Label smoothing was introduced by Szegedy et al. (2016) in the context of image classification, but has since been shown to provide performance improvements in other areas including speech recognition (Chorowski & Jaitly 2016), machine translation (Vaswani et al. 2017) and language modeling (Pereyra et al. 2017). The precise mechanism by which label smoothing improves test performance is not well understood although Müller et al. (2019) show that it improves the calibration of the predicted output probabilities. A closely related technique is *DisturbLabel* (Xie et al. 2016), in which a certain percentage of the labels in each batch are randomly switched at each iteration of training.

Bayesian approaches: For some models, including the simplified neural network model in figure 9.11, the Bayesian predictive distribution can be computed in closed form. This is explained in detail in Bishop (2006) and Prince (2012). For neural networks, the posterior distribution over the parameters cannot be represented in closed form and must be approximated. The two main approaches are variational Bayes (Hinton & van Camp 1993; MacKay et al. 1995; Barber & Bishop 1997; Blundell et al. 2015), in which the posterior is approximated by a simpler tractable distribution, and Markov Chain Monte Carlo (MCMC) methods, which approximate the distribution by drawing a set of samples (Neal 1995; Welling & Teh 2011; Chen et al. 2014b; Ma et al. 2015; Li et al. 2016a). The generation of samples can be integrated into the stochastic gradient descent algorithm, and this is known as stochastic gradient MCMC (see Ma et al. 2015). It has recently been discovered that "cooling" the posterior distribution over the parameters

(making it sharper) improves the predictions from these models (Wenzel et al. 2020a), but this is not currently fully understood (see Noci et al. 2021).

Transfer learning: Transfer learning for visual tasks works extremely well (Sharif Razavian et al. 2014) and has supported rapid progress in computer vision including the original AlexNet results (Krizhevsky et al. 2012). Transfer learning has also had an enormous impact on natural language processing where many models are based on pre-trained features from the BERT language model (Devlin et al. 2018). More information about transfer learning can be found in Zhuang et al. (2020) and Yang et al. (2020b).

Self-supervised llearning Self-supervised learning techniques for images have included inpainting masked image regions (Pathak et al. 2016), predicting the relative position of patches in an image (Doersch et al. 2015), re-arranging permuted image tiles back into their original configuration (Noroozi & Favaro 2016), colorizing grayscale images (Zhang et al. 2016c), and transforming rotated images back to their original orientation (Gidaris et al. 2018). In SimCLR (Chen et al. 2020c), a network is learned that maps versions of the same image that have been photometrically and geometrically transformed to the same representation, while repelling versions of different images, with the goal of becoming indifferent to irrelevant image transformations. Jing & Tian (2020) present a survey of self-supervised learning in images.

Self-supervised learning in natural language processing could be based on masking words from a sentence (Devlin et al. 2018), predicting the next word in a sentence (Radford et al. 2019; Brown et al. 2020), or predicting whether two sentences follow one another (Devlin et al. 2018). In the field of automatic speech recognition, the Wav2Vec model (Schneider et al. 2019) aims to distinguish an original audio sample from one where 10ms of audio has been swapped out from elsewhere in the clip. Self-supervision has also been applied to graph neural networks (chapter 13) and tasks include recovering masked features (You et al. 2020) and recovering the adjacency structure of the graph (Kipf & Welling 2016b). A review of graph self-supervised learning can be found in Liu et al. (2021b).

Data augmentation Data augmentation for images dates back to at least LeCun et al. (1998a), and contributed to the success of AlexNet (Krizhevsky et al. 2012), in which the dataset was increased by a factor of 2048. Typical augmentation approaches for images include geometric transformations, changing or manipulating the color space, noise injection, and applying spatial filters. More elaborate techniques include randomly mixing images (Inoue 2018; Summers & Dinneen 2019), randomly erasing parts of the image (Zhong et al. 2020), style transfer (Jackson et al. 2019), and randomly swapping image patches (Kang et al. 2017). In addition, many studies have used generative adversarial networks or GANs (see chapter 14) to produce novel, but plausible data examples (e.g., Calimeri et al. 2017). In other cases, the data have been augmented with adversarial examples (Goodfellow et al. 2014), which are minor perturbations of the training data that cause the example to be misclassified. A comprehensive review of data augmentation for images can be found in Shorten & Khoshgoftaar (2019).

Augmentation methods for acoustic data include pitch shifting, time stretching, dynamic range compression, and adding random noise (e.g., Abeßer et al. 2017; Salamon & Bello 2017; Xu et al. 2015; Lasseck 2018), as well as mixing data pairs (Zhang et al. 2017; Yun et al. 2019), masking features (Park et al. 2019), and using GANs to generate new data (Mun et al. 2017). Augmentation for speech data includes vocal tract length perturbation (Jaitly & Hinton 2013; Kanda et al. 2013), style transfer (Gales 1998; Ye & Young 2004), adding noise (Hannun et al. 2014), and synthesizing speech (Gales et al. 2009).

Augmentation methods for text data include adding noise at a character level by switching, deleting, and inserting letters (Belinkov & Bisk 2017; Feng et al. 2020), or by generating adversarial examples (Ebrahimi et al. 2017), using common spelling mistakes (Coulombe 2018), randomly swapping or deleting words (Wei & Zou 2019), using synonyms (Kolomiyets et al.

2011), altering adjectives (Li et al. 2017), passivization (Min et al. 2020), using generative models to create new data (Qiu et al. 2020), and round-trip translation to another language and back again (Aiken & Park 2010). A review of augmentation methods for text data can be found in Bayer et al. (2021).

Problems

Problem 9.1 Consider a model where the prior distribution over the parameters is a normal distribution with mean zero and variance $\sigma_{\boldsymbol{\phi}}^2$ so that

$$Pr(\boldsymbol{\phi}) = \prod_{j=1}^{J} \text{Norm}_{\phi_j}[0, \sigma_{\boldsymbol{\phi}}^2]. \tag{9.21}$$

where j indexes the model parameters. When we apply a prior, we maximize $\prod_{i=1}^{I} Pr(\mathbf{x}_i|\boldsymbol{\phi})Pr(\boldsymbol{\phi})$. Show that the associated loss function of this model is equivalent to L2 regularization.

Problem 9.2 How do the gradients in the backpropagation algorithm (section 7.2.2) change when L2 regularization (equation 9.5) is added?

Problem 9.3 Consider a univariate linear regression problem $y = \phi_0 + \phi_1 x$ where x is the input, y is the output, and ϕ_0 and ϕ_1 are the intercept and slope parameters respectively. Assume that we have I data examples $\{x_i, y_i\}$ and are using a least squares loss function. Consider adding Gaussian noise with mean 0 and variance σ_x^2 to the inputs x_i during each iteration of training. What is the expected gradient update?

Problem 9.4 Derive the loss function for multi-class classification when we use label smoothing so that the target probability distribution has 0.9 at the correct class and the remaining probability mass of 0.1 is divided between the remaining $D_o - 1$ classes.

Problem 9.5 Show that the weight decay parameter update with decay rate λ':

$$\boldsymbol{\phi} \longleftarrow (1 - \lambda')\boldsymbol{\phi} - \alpha \frac{\partial L}{\partial \boldsymbol{\phi}}, \tag{9.22}$$

on the original loss function $\text{L}[\boldsymbol{\phi}]$ is equivalent to a standard gradient update using L2 regularization so that the modified loss function $\tilde{\text{L}}[\boldsymbol{\phi}]$ is:

$$\tilde{\text{L}}[\boldsymbol{\phi}] = \text{L}[\boldsymbol{\phi}] + \frac{\lambda'}{2\alpha} \sum_k \phi_k^2, \tag{9.23}$$

where $\boldsymbol{\phi}$ are the parameters, and α is the learning rate.

Problem 9.6 Consider a model with two parameters $\boldsymbol{\phi} = [\phi_0, \phi_1]^T$. Draw the $L0$, $L\frac{1}{2}$, $L1$, and $L2$ regularization terms in a similar form to that shown in figure 9.1b.

Chapter 10

Convolutional networks

Chapters 2-9 introduced the supervised learning pipeline for deep neural networks. However, only fully connected networks with a single sequential path from input to output were considered. Chapters 10-13 introduce more specialized network components with sparser connections, shared weights, and parallel processing paths. This chapter describes *convolutional layers*, which are suited to processing image data.

Images have three properties that suggest the need for specialized architecture. First, they are high-dimensional. A typical image for a classification task contains 224×224 RGB values (i.e., 150,528 input dimensions). Hidden layers in fully connected networks are generally larger than the input size, and so even for a shallow network, the number of weights would exceed $150,528^2$ or 22 billion. This poses obvious practical problems in terms of the required training data, memory, and computation.

Second, nearby image pixels are statistically related. However, a fully connected network has no notion of "nearby" and treats the relationship between every input equally; if the pixels of the training and test images were randomly permuted in the same way, the network could still be trained with no practical difference. Third, the interpretation of an image is stable under geometric transformations. An image of a tree is still an image of a tree if we shift it leftwards by a few pixels. However, this shift changes every input to the network, and so the model would have to learn the patterns of pixels that correspond to a tree separately at every position. This is clearly inefficient.

These properties lead to the idea of sharing the parameters that process each local region of the image; we will need to learn fewer parameters, can exploit the local statistics, and do not have to re-learn the interpretation of the pixels at every position. Convolutional layers share parameters in this way. A network that predominantly consists of convolutional layers is known as a *convolutional neural network* or *CNN*.

10.1 Invariance and equivariance

We argued above that certain properties of images are stable under transformations. In this section, we make this idea more mathematically precise. A function f[$\mathbf{x}$] of an image

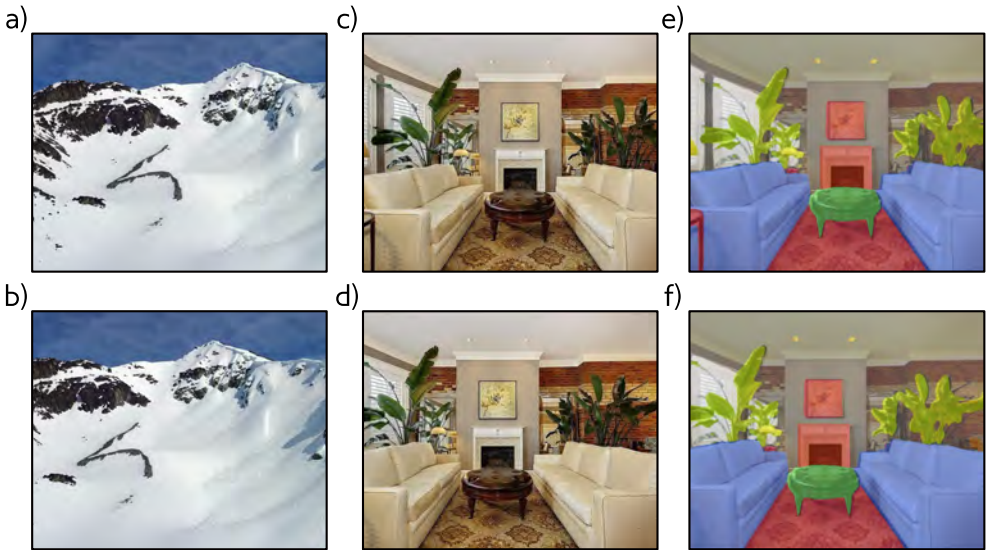

Figure 10.1 Invariance and equivariance for translation. a–b) In image classification the goal is to categorize both of these images as "mountain" regardless of the horizontal shift that has occurred. In other words, we require the network prediction to be invariant to translation. c–d) In a semantic segmentation task, the goal is to associate a label with each pixel. e–f) When the input image is translated we want the output (colored overlay) to translate accordingly. In other words, we require the output to be equivariant with respect to translation. Panels c–f) adapted from Bousselham et al. (2021)

$\mathbf{x}$ is *invariant* to a transformation $\mathbf{t}[\mathbf{x}]$ if:

$$\mathbf{f}[\mathbf{t}[\mathbf{x}]] = \mathbf{f}[\mathbf{x}]. \tag{10.1}$$

In other words, the output of the function $\mathrm{f}[\mathbf{x}]$ is the same regardless of the transformation $\mathbf{t}[\mathbf{x}]$. Networks for image classification should be invariant to geometric transformations of the image (figure 10.1a–b). The network $\mathbf{f}[\mathbf{x}]$ should identify an image as containing the same object, even if it has been translated, rotated, flipped, or warped.

A function $\mathrm{f}[\mathbf{x}]$ of an image $\mathbf{x}$ is *equivariant* to a transformation $\mathbf{t}[\mathbf{x}]$ if:

$$\mathbf{f}[\mathbf{t}[\mathbf{x}]] = \mathbf{t}[\mathbf{f}[\mathbf{x}]]. \tag{10.2}$$

In other words, $\mathbf{f}[\mathbf{x}]$ is equivariant to the transformation $\mathbf{t}[\mathbf{x}]$ if its output changes in the same way under the transformation as the input. Networks for per-pixel image segmentation should be equivariant to transformations (figure 10.1c-f); if the image is translated, rotated, or flipped, the network $\mathbf{f}[\mathbf{x}]$ should return a segmentation that has been transformed in the same way.

10.1.1 Invariance from augmentation

Section 9.3.8 described augmentation, in which the training dataset is expanded by transforming each example in multiple ways. In principle, this could be used to induce invariance in image classification by transforming the data $\mathbf{x}$ without changing the associated label y. Similarly, augmentation could also be used to induce equivariance in semantic segmentation by transforming the data $\mathbf{x}$ and transforming the label map $\mathbf{y}$ in the same way. These are viable strategies, but it would be preferable to design the network architecture to have these properties rather than use this brute-force approach.

10.2 Convolutional networks for 1D inputs

Convolutional networks consist of a series of convolutional layers, each of which is equivariant to translation. They also typically include pooling mechanisms that induce partial invariance to translation. For clarity of exposition, we first consider convolutional networks for 1D data, which are easier to visualize. In section 10.3 we progress to 2D convolution, which can be applied to image data.

10.2.1 Convolution for 1D inputs

Convolution in 1D is an operation applied to a vector $\mathbf{x} = [x_1, x_2, \ldots, x_I]$ of inputs. Each output z_i is a weighted sum of nearby inputs. The weights are the same at every position i and are called a kernel or *filter*. The size of the region over which inputs are weighted and summed is termed the *kernel size*. For a kernel size of three we have:

$$\begin{aligned} z_i &= \omega_1 x_{i-1} + \omega_2 x_i + \omega_3 x_{i+1} \\ &= \sum_{j=1}^{3} \omega_j x_{i+j-1}, \end{aligned} \tag{10.3}$$

where $\boldsymbol{\omega} = [\omega_1, \omega_2, \omega_3]^T$ is the kernel (figure 10.2). Notice that the convolution operation is equivariant with respect to translation.[1] If we translate the input, then the corresponding output is translated in the same way.

Problem 10.1

10.2.2 Padding

Equation 10.4 shows that each output is computed by taking a weighted sum of the previous, current, and subsequent positions in the input. This begs the question of how

[1] Strictly speaking, this is a cross-correlation and not a convolution, in which the weights would be flipped relative to the input (so we would replace x_{i+j-1} with x_{i-j+1}). Regardless, this (incorrect) definition is the usual convention in machine learning.

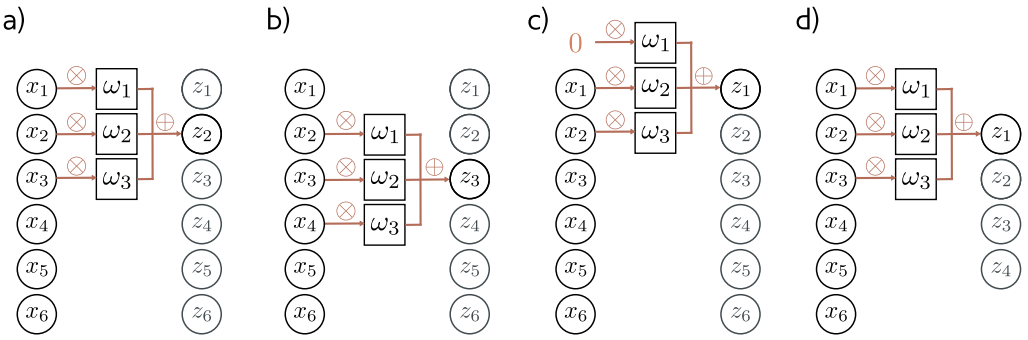

Figure 10.2 1D convolution with kernel size three. Each output z_i is a weighted sum of the nearest three inputs x_{i-1}, x_i, and x_{i+1}, where the weights are $\boldsymbol{\omega} = [\omega_1, \omega_2, \omega_3]$. a) Output z_2 is computed as $z_2 = \omega_1 x_1 + \omega_2 x_2 + \omega_3 x_3$. b) Output z_3 is computed as $z_3 = \omega_1 x_2 + \omega_2 x_3 + \omega_3 x_4$. c) At position z_1, the kernel extends beyond the first input x_1. This can be handled by zero padding, in which we assume values outside the input are zero. The last output is treated similarly. d) Alternatively, we could only compute outputs where the kernel fits within the input range ("valid" convolution); now the output will be smaller than the input.

to deal with the first output (where there is no previous input) and the last output (where the is no subsequent input).

There are two common approaches. The first is to pad the edges of the inputs with new values and then proceed as normal. *Zero padding* assumes that the input is zero outside its valid range (figure 10.2c). Other possibilities include treating the input as circular or reflecting it at the boundaries. The second approach is to discard the output positions where the kernel exceeds the range of input positions. These *valid convolutions* have the advantage that no extra information is introduced at the edges of the input. However, they have the disadvantage that the representation decreases in size.

10.2.3 Stride, kernel size, and dilation

In the example above, each output was a sum of the nearest three inputs. However, this is just one of a larger family of convolution operations, the members of which are distinguished by their stride, kernel size, and dilation rate. When we evaluate the output at every position, we term this a *stride* of one. However, it is also possible to shift the kernel by a stride of greater than one. If we have a stride of two, then we create roughly half the number of outputs (figure 10.3a–b).

To integrate over a larger area, the *kernel size* can be increased (figure 10.3c). This typically remains an odd number so that it can be centered around the current position. Increasing the kernel size has the disadvantage of requiring more weights. This leads to the idea of *dilated* or *atrous* convolutions, in which the kernel values are interspersed

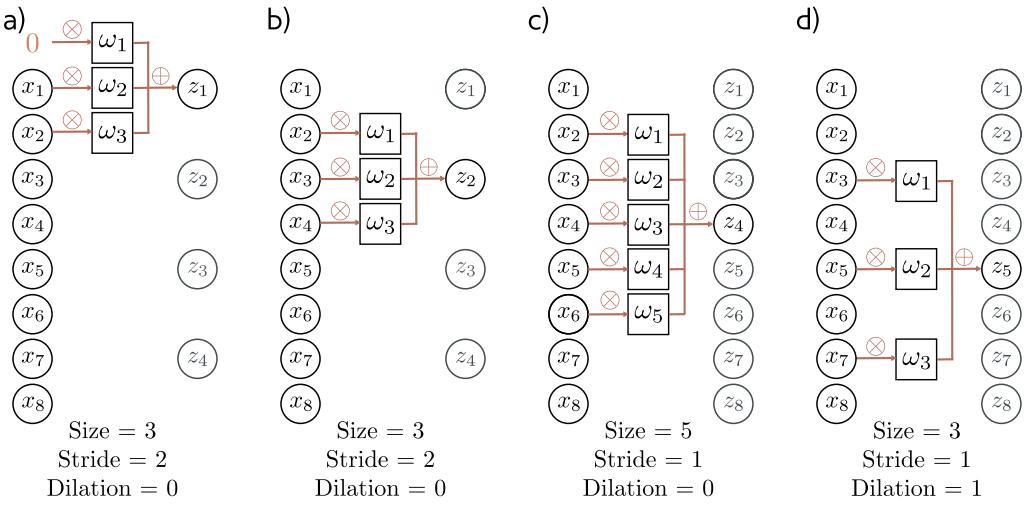

Figure 10.3 Stride, kernel size, and dilation. a) With a stride of two we evaluate the kernel at every other position and so the first output z_1 is computed from a weighted sum centered at x_1 and b) the second output z_2 is computed from a weighted sum centered at x_3 and so on. c) The kernel size can also be changed. With a kernel size of five we take a weighted sum of the nearest five inputs. d) In dilated or atrous convolution, we intersperse zeros in the weight vector to allow us to integrate over a larger area while still using fewer weights.

with zeros. For example, we can turn a kernel of size five into a dilated kernel of size three by setting the second and fourth elements to zero. We still integrate information from a larger region of the input but only require three weights to do this (figure 10.3d). The number of zeros that we intersperse between the weights is termed the *dilation rate*.

Problems 10.2-10.4

10.2.4 Convolutional layers

A convolutional layer computes its output by convolving the input, adding a bias β, and passing each result through an activation function $\mathrm{a}[\bullet]$. With kernel size three, stride one, and dilation rate zero, the i^{th} hidden unit h_i would be computed as:

$$\begin{aligned} h_i &= \mathrm{a}\left[\beta + \omega_1 x_{i-1} + \omega_2 x_i + \omega_3 x_{i+1}\right] \\ &= \mathrm{a}\left[\beta + \sum_{j=1}^{3} \omega_j x_{i+j-1}\right], \end{aligned} \tag{10.4}$$

where the bias β and kernel weights $\omega_1, \omega_2, \omega_3$ are trainable parameters. This is a special case of a fully connected layer that computes the i^{th} hidden unit as:

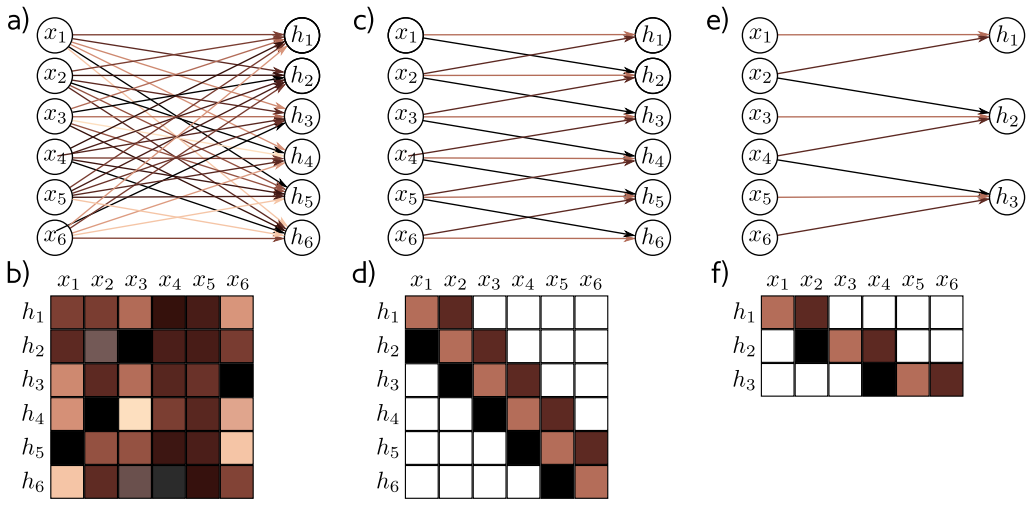

Figure 10.4 Fully connected vs. convolutional layers. a) A fully connected layer has a weight connecting each input x to each hidden unit h (colored arrows) and a bias for each hidden unit (not shown). b) The associated weight matrix $\boldsymbol{\Omega}$ contains 36 weights relating the six inputs to the six hidden units. c) A convolutional layer with kernel size three computes each hidden unit as the same weighted sum of the three neighboring inputs (arrows) plus a bias (not shown). d) The weight matrix is a special case of the fully connected matrix where many weights are zero and others are repeated (same colors indicate same value, white indicates zero weight). e) A convolutional layer with kernel size three and stride two computes a weighted sum at every other position. f) This is also a special case of a fully connected network but with a different sparse weight structure.

$$h_i = \mathrm{a}\left[\beta_i + \sum_{j=1}^{D} \omega_{ij} x_j\right]. \tag{10.5}$$

If there are D inputs $x_\bullet$ and D hidden units $h_\bullet$, then this fully connected layer would have D^2 weights $\omega_{\bullet\bullet}$ and D biases $\beta_\bullet$. The convolutional layer only uses three weights and one bias. This can be exactly reproduced by the fully connected layer if most weights are set to zero and others are constrained to be identical (figure 10.4).

Problem 10.5

10.2.5 Channels

If we only apply a single convolution, then information will inevitably be lost; we are averaging nearby inputs and the ReLU activation function clips results that are less than

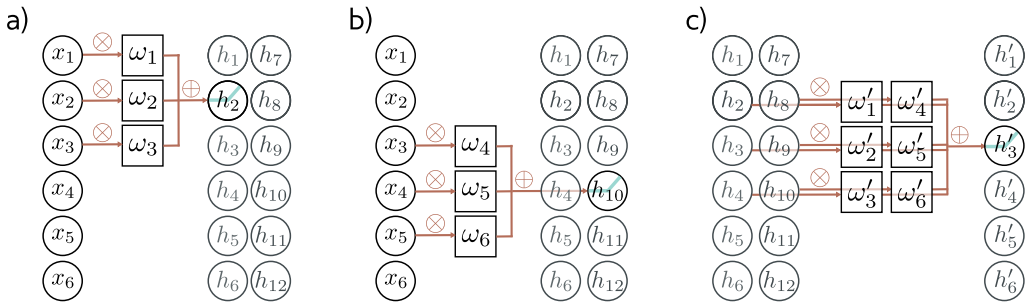

Figure 10.5 Channels. Typically multiple convolutions are applied to the input $\mathbf{x}$ and stored in channels. a) A convolution is applied to create hidden units h_1 to h_8, which form the first channel. b) A second convolution operation is applied to create hidden units h_9 to h_{16}, which form the second channel. The channels are stored in a 2D array $\mathbf{H}_1$ that contains all the hidden units in the first hidden layer. c) If we add a second convolutional layer, then there is now one hidden unit per channel at each input position. The convolution kernel now defines a weighted sum over all of the input channels at the three closest positions to create each new output channel. If the input has C_i channels and the kernel size is K, then there will be $C_i \times K$ weights and one bias needed to create each of the C_o output channels. This makes a total of $C_i \times C_o \times K$ weights and C_o biases.

zero. Hence, it is usual to compute several convolutions in parallel. Each convolution produces a new set of hidden variables, which we term a *feature map* or *channel*.

Figure 10.5a–b illustrate this with two convolution kernels of size three and zero padding. The first kernel computes a weighted sum of the nearest three pixels, adds a bias, and passes the results through the activation function to produce hidden units h_1 to h_8. These comprise the first channel. The second kernel computes a different weighted sum of the nearest three pixels, adds a different bias, and passes the results through the activation function to create hidden units h_9 to h_{16}. These comprise the second channel.

Problem 10.6

The input and all subsequent hidden layers may have multiple channels (figure 10.5c). If the incoming layer has C_i channels and the kernel size is K, then the hidden units in each output channel are computed as a weighted sum over all C_i channels and K kernel positions using the weight matrix $\boldsymbol{\Omega} \in \mathbb{R}^{C_i \times K}$ and one bias. Hence, if there are C_o channels in the next layer, then we need $\boldsymbol{\Omega} \in \mathbb{R}^{C_i \times C_o \times K}$ weights and $\boldsymbol{\beta} \in \mathbb{R}^{C_o}$ biases.

Problem 10.7

Problem 10.8

10.2.6 Convolutional networks and receptive fields

A convolutional network consists of a series of convolutional layers. The *receptive field* of a hidden unit in the network is the region of the original input that feeds into it. Consider a convolutional network where each convolutional layer has kernel size three. The hidden units in the first layer take a weighted sum of the three closest input positions, and so have receptive fields of size three. The hidden units in the second layer take a weighted sum

of the three closest positions in the first layer. However, these are themselves weighted sums of three inputs. Hence, the hidden units in the second layer have a receptive field of size five. In this way, the receptive field of hidden units in successive layers increases, and information from across the input is gradually integrated (figure 10.6).

Problem 10.9

10.2.7 Example: MNIST1D

Let's apply a small convolutional network to the MNIST1D data (figure 8.1). The input $\mathbf{x}$ is a 40D vector, and the output $\mathbf{f}$ is a 10D vector that is passed through a softmax layer to produce class probabilities. We use a network with three hidden layers (figure 10.7). The fifteen channels of the first hidden layer are computed using a kernel size of three and a stride of two with "valid" padding. This creates a hidden layer $\mathbf{H}_1$ containing nineteen spatial positions and fifteen channels.

The second hidden layer $\mathbf{H}_2$ is also computed using a kernel size of three, a stride of two, and "valid" padding The third hidden layer is computed similarly. At this stage, the representation has only four spatial positions and fifteen channels. These values are reshaped into a 1D vector of size sixty. Finally, a fully connected layer maps sixty values to the ten output activations. The total number of parameters in this network is 2050.

This network was trained for 100,000 steps using SGD with a learning rate of 0.01 and a batch size of 100 on a dataset of 4,000 examples. We compare this with a fully connected network with the same number of layers and hidden units (i.e., three hidden layers with 285, 135, and 60 hidden units, respectively, making a total of 150,185 parameters). By the logic of figure 10.4, the convolutional network is a special case of the fully connected one; the latter has enough flexibility to exactly replicate the former. Figure 10.8 shows that both models are flexible enough to fit the training data perfectly. However, the test error for the convolutional network is much less than for the fully connected network.

Problem 10.10

We know that over-parameterized networks work well (section 8.4.1), and so the difference in the number of parameters is probably not the reason for this discrepancy. The explanation is that convolutional architecture has a superior inductive bias (i.e., interpolates to new points better) because we have embodied some prior knowledge in the architecture; we have forced the network to process each position in the input in the same way. We know that the data were created by starting with a template that is (among other operations) randomly translated, and so this is sensible.

The fully connected network has to learn what each digit template looks like at every position, whereas the convolutional network shares information across positions and hence learns to identify each category more accurately. Another way of thinking about this is that when we train the convolutional network, we search through a smaller family of input/output mappings, all of which are plausible. When we train the fully connected network, we are searching through a much larger family.

This is a major advantage of deep neural networks; there is no easy way to incorporate prior knowledge about translational equivariance into a network with a single hidden layer. With enough hidden units, the shallow network could reproduce the same function, but it might take a great deal more training data to achieve the same results.

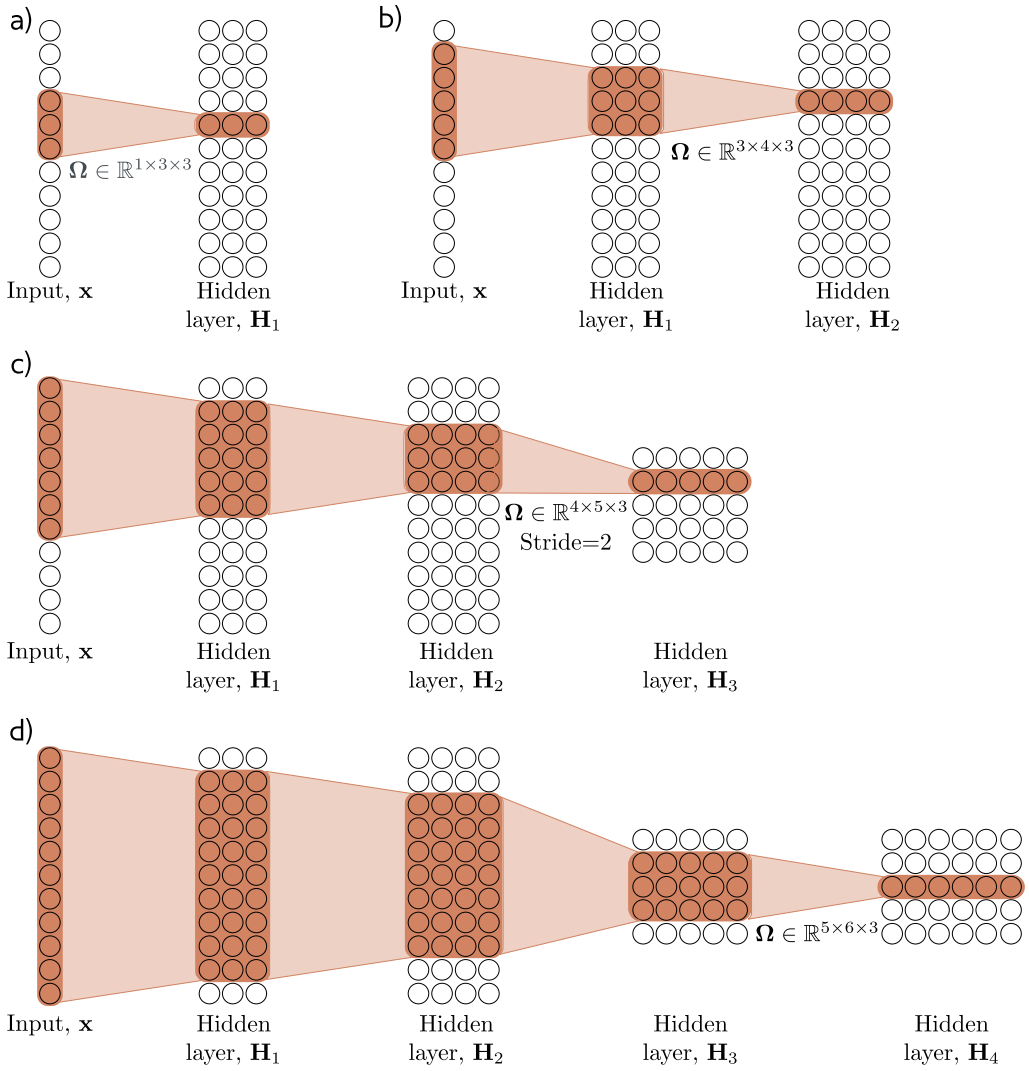

Figure 10.6 Receptive fields for network with kernel width of three. a) An input with eleven dimensions feeds into a hidden layer with three channels and convolution kernel of size three. The pre-activations of the three highlighted hidden units in the first hidden layer $\mathbf{H}_1$ are different weighted sums of the nearest three inputs and so the receptive field in $\mathbf{H}_1$ has size three. b) The pre-activations of the four highlighted hidden units in layer $\mathbf{H}_2$ each take a weighted sum of the three channels in layer $\mathbf{H}_1$ at each of the three nearest positions. Each of the hidden units in layer $\mathbf{H}_1$ weights the nearest three positions of the input. Hence, hidden units in $\mathbf{H}_2$ have a receptive field size of five. c) Adding a third layer with kernel of size three, and stride two increases the receptive field size further to seven. d) By the time that we add a fourth layer the receptive field of the hidden units at position three has grown to cover the entire input.

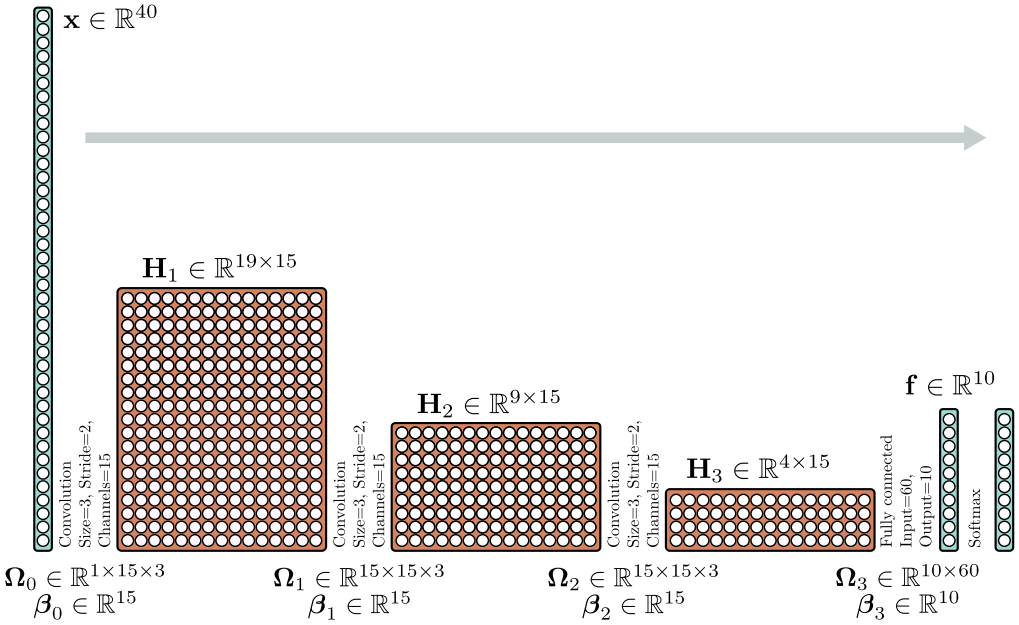

Figure 10.7 Convolutional network for classifying MNIST1D data (see figure 8.1). The MNIST1D input has dimension $D_i = 40$. The first convolutional layer has fifteen channels, kernel size three, stride two, and only retains "valid" positions to make a representation with nineteen positions and fifteen channels. The next two convolutional layers have the same settings, which gradually reduces the representation size. Finally, a fully connected layer takes all sixty hidden units from the third hidden layer and outputs ten activations that are subsequently passed through a softmax layer to produce the ten class probabilities.

10.3 Convolutional networks for 2D inputs

The previous section described a convolutional network for processing 1D data. Such networks have been applied to data such as financial time-series, audio, and text. However, convolutional networks are more usually applied to 2D image data. The convolutional kernel now becomes a 2D object. A 3×3 kernel $\boldsymbol{\Omega} \in \mathbb{R}^{3\times 3}$ applied to a single channel 2D input $\mathbf{X}$ would compute a single layer of hidden units $\mathbf{H}$ as:

$$h_{i,j} = \mathrm{a}\left[\beta + \sum_{m=1}^{3}\sum_{n=1}^{3}\omega_{m,n}x_{i+m-1,j+n-1}\right], \tag{10.6}$$

where $\omega_{\bullet,\bullet}$ are the entries of the convolutional kernel. Now we are simply computing a weighted sum over a square 3×3 region of the input and translating the kernel both horizontally and vertically across the 2D input (figure 10.9).

Problem 10.11

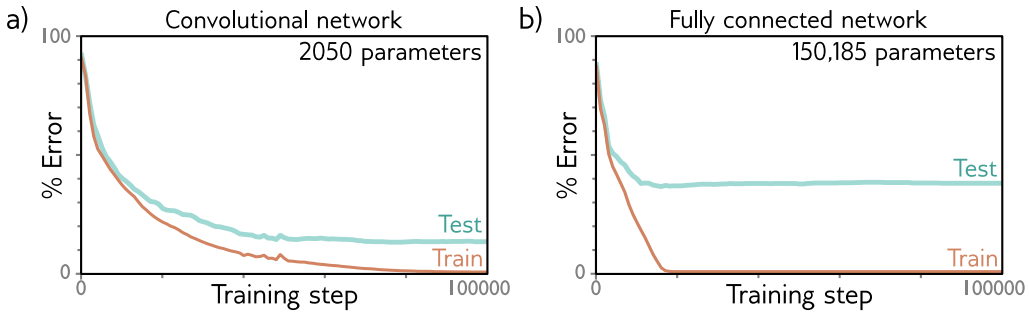

Figure 10.8 MNIST1D results. a) The convolutional network from figure 10.7 eventually fits the training data perfectly, and the performance on the test set levels out at around a 17% error rate. b) We compare this with a fully connected network with the same number of hidden layers and the same number of hidden units in each. This much larger model learns the training data more quickly but fails to generalize well with an aysmptotic test error rate of 40%. The fully connected model can exactly reproduce the convolutional model but fails to do so; the convolutional structure reduces the family of models to those that process every position similarly, and this restriction improves model performance.

Often the input is an RGB image, and this is treated as a 2D signal with three channels (figure 10.10). Here, a 3×3 kernel would have 3×3×3 weights and be applied to the three input channels at each of the 3×3 positions. To generate multiple output channels, we repeat this process with different kernel weights and stack the results together to form a 3D tensor. If the kernel is size $K \times K$, and there are C_i input channels, each output channel is a weighted sum of $K \times K \times C_i$ quantities and one bias. It follows that to compute C_o output channels, we need $K \times K \times C_i \times C_o$ weights and C_o biases.

10.4 Downsampling and upsampling

The MNIST1D network (figure 10.7) increased its receptive field size by steadily scaling down the size of the representation at each layer using convolutions with a stride of two. In this section, we consider other methods for scaling down or *downsampling* the representation when the input is 2D. We also consider methods for scaling them back up (*upsampling*). This is useful when the output is the same size as the input; here, it is typical to downsample the representation to integrate information from across the image, then upsample it again to produce the output. Finally, we consider methods to change the number of channels between layers. This is used when we need to combine layers with different numbers of channels from two branches of a network.

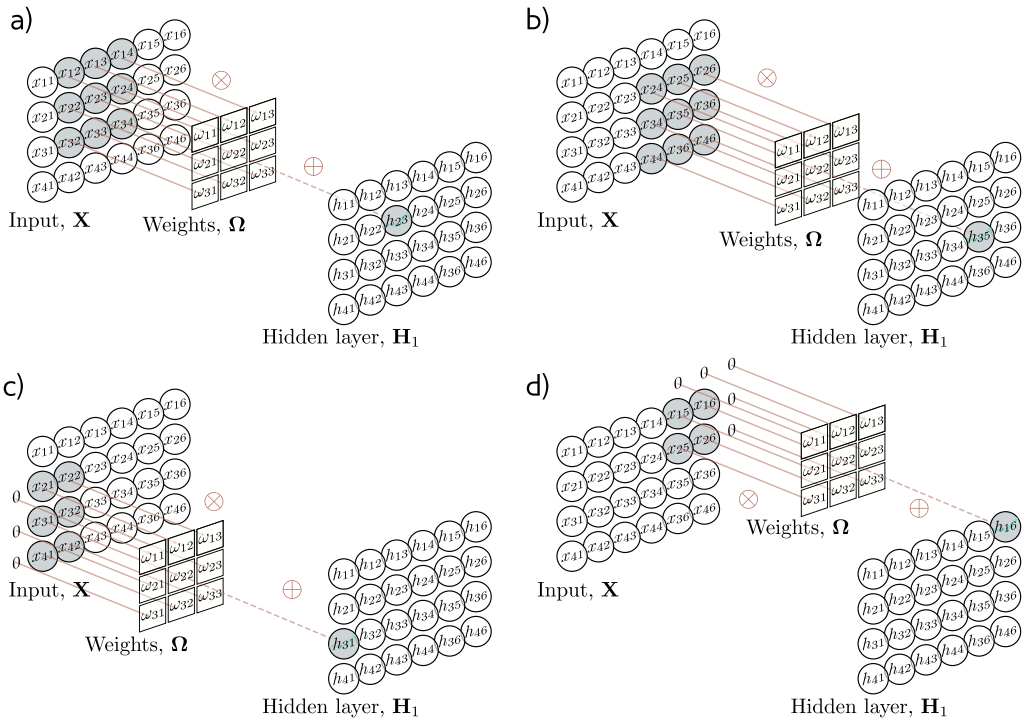

Figure 10.9 2D convolutional layer. Each output h_{ij} computes a weighted sum of the 3×3 nearest inputs, adds a bias, and passes the result through an activation function. a) Here we are using a stride of one, and so the output h_{23} (shaded output) is a weighted sum of the nine positions from x_{12} to x_{34} (shaded inputs). b) Different outputs are computed by translating the kernel across the image grid in two dimensions. c–d) With zero-padding, positions beyond the edge of the image are considered to be zero.

10.4.1 Downsampling

There are three main approaches to scaling down a 2D representation. Here, we consider the most common case of scaling down both dimensions by a factor of two. The first approach is to sample every other position. When we use a stride of two, we are effectively applying this method simultaneously with the convolution operation (figure 10.11a).

Problem 10.12

Max-pooling retains the maximum over the 2×2 input values (figure 10.12b) and induces some invariance to translation; if the input is shifted by a single pixel, then many of these maximum values will remain the same. *Mean pooling* or *average pooling* averages the 2×2 input values. For all three approaches, the downsampling is applied separately within each channel, so the output representation will have half the width and half the height, but the same number of channels.

Problem 10.13

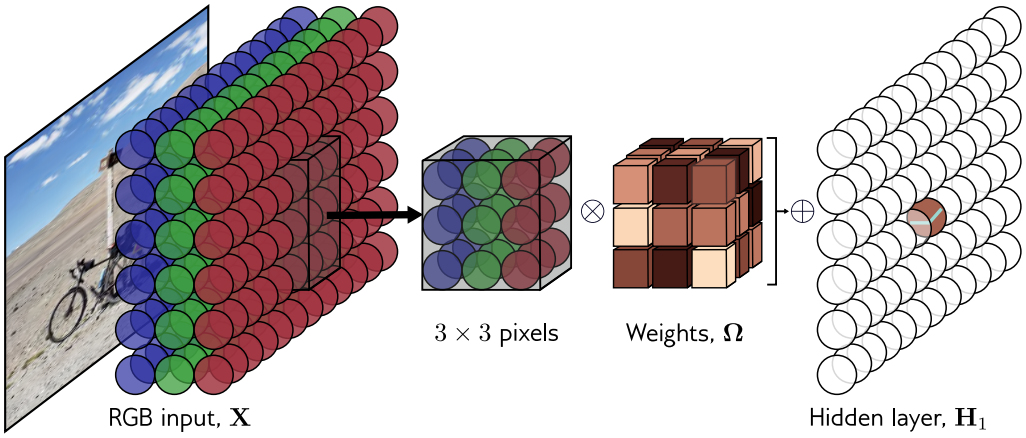

Figure 10.10 2D convolution applied to an image. The image is treated as a 2D input with three channels corresponding to the red, green, and blue components. With a 3×3 kernel, each pre-activation in the first hidden layer is computed by pointwise multiplying the 3×3×3 kernel weights with the 3×3 RGB image patch centered at the same position, summing, and adding the bias. To compute all of the pre-activations in the hidden layer, we "slide" the kernel over the image in both horizontal and vertical directions. The output is a 2D layer of hidden units. To create multiple output channels, we would repeat this process with multiple kernels, resulting in a 3D array of hidden units at hidden layer $\mathbf{H}_1$

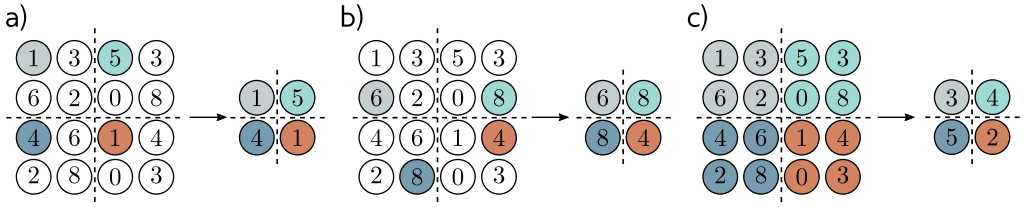

Figure 10.11 Methods for scaling down representation size (downsampling). a) Sub-sampling. The original 4×4 representation (left) is reduced to size 2×2 (right) by retaining every other input. Colors on the left indicate which inputs contribute to the outputs on the right. This is effectively what happens with a kernel of stride two except that the intermediate values are never computed. b) Max-pooling. Each output comprises the maximum value of the corresponding 2×2 block. c) Mean-pooling. Each output is the mean of the values in the 2×2 block.

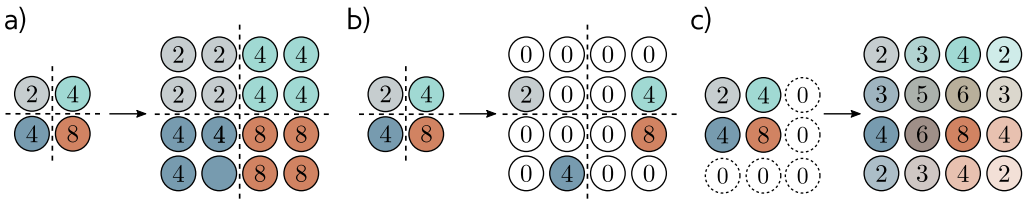

Figure 10.12 Methods for scaling up representation size (upsampling). a) The simplest way to double the size of a 2D layer is to duplicate each input four times. b) In networks where we have previously used a max-pooling operation (figure 10.11b), we can redistribute the values to the same positions that they originally came from (i.e., where the maxima were). This is known as max-unpooling. c) A third option is to bilinearly interpolate between the input values.

10.4.2 Upsampling

The simplest way to scale up a network layer to double the resolution is to duplicate all of the channels at each spatial position four times (figure 10.12a). A second method is max-unpooling; this is used where we have previously used a max-pooling operation for downsampling and we distribute the values to the positions that they originated from (figure 10.12b). A second approach uses bilinear interpolation to fill in the missing values between the points where we have samples. (figure 10.12c).

Problem 10.14

A fourth approach is roughly analogous to downsampling using a stride of two. In that method, there were half as many outputs as inputs and for kernel size three each output was a weighted sum of the three closest inputs (figure 10.13a). In *transposed convolution*, this picture is reversed (figure 10.13c). There are twice as many outputs as inputs and each input contributes to three of the outputs When we consider the associated weight matrix of this upsampling mechanism (figure 10.13d), we see that it is the transpose of the matrix for the downsampling mechanism (figure 10.13b).

10.4.3 Changing the number of channels

Sometimes we want to change the number of channels between one hidden layer and the next, without further spatial pooling. This might be so that we can combine the hidden layer with another computation that has been done in parallel (see chapter 11). To accomplish this, we apply a convolution with a kernel size of one. Each element of the output layer is now computed by taking a weighted sum of all of the channels at the current position (figure 10.14). We can repeat this multiple times with different weights to generate as many output channels as we need. The associated convolution weights have size $1 \times 1 \times C_i \times C_o$. Hence, this is known as *1×1 convolution*. When combined with a bias and activation function, it is equivalent to running the same fully connected network on the channels at every position.

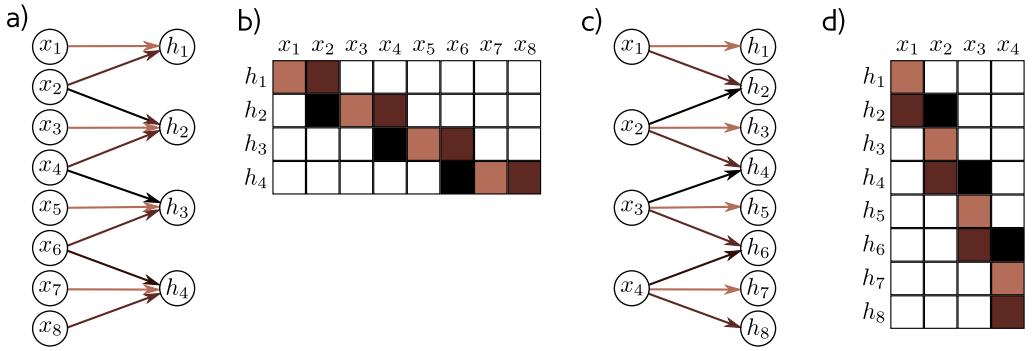

Figure 10.13 Transposed convolution in 1D. a) Downsampling with kernel size three, stride two, and zero padding. Each output is a weighted sum of three inputs (arrows indicate weights). b) This can be expressed by a weight matrix (same color indicates shared weight). c) In transposed convolution, each input contributes three values to the output layer, which has twice as many outputs as inputs. d) The associated weight matrix is the transpose of that in panel (b).

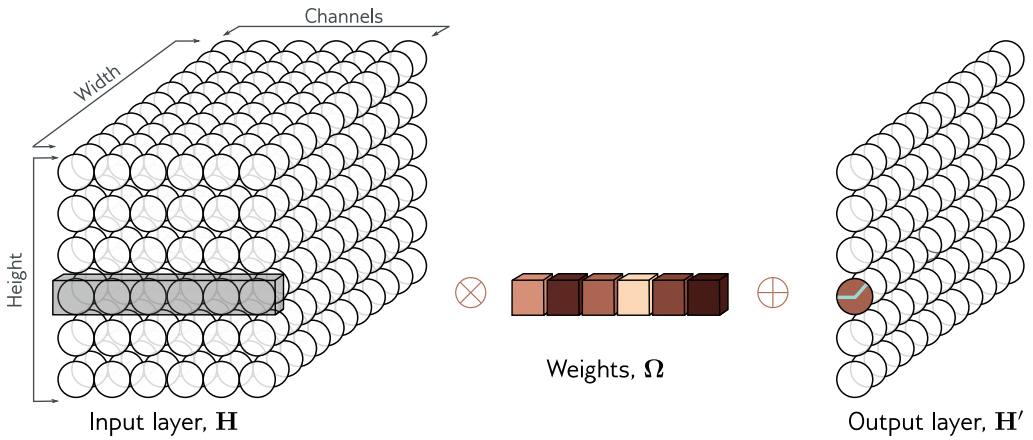

Figure 10.14 1×1 convolution. To change the number of channels without spatial pooling, we apply a 1×1 kernel. In practice, this means that each output channel is computed by taking a weighted sum of all of the channels at the same position, adding a bias, and passing through an activation function. Multiple output channels are created by repeating this operation with different weights and biases.

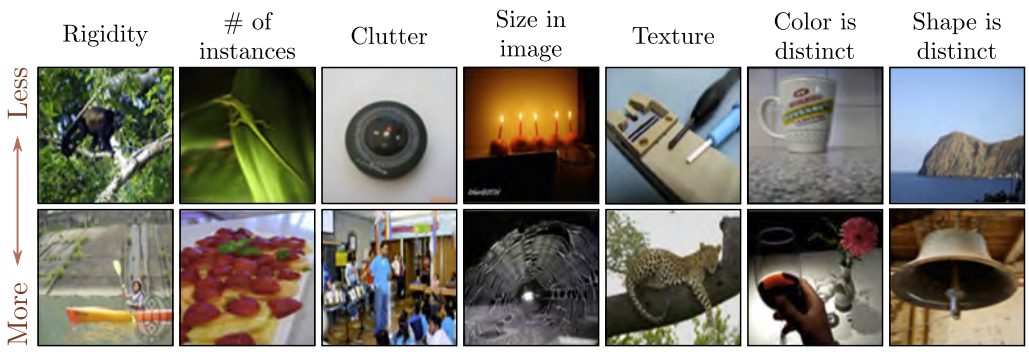

Figure 10.15 ImageNet classification task. The model must classify an input image into one of 1000 classes. The images vary greatly along a wide range of attributes (columns). These include rigidity (monkey < canoe), number of instances in image (lizard < strawberry), clutter (compass < steel drum), size in image (candle < spiderweb), texture (screwdriver < leopard), distinctiveness of color (mug < red wine), and distinctiveness of shape (headland < bell). Adapted from Russakovsky et al. (2015).

10.5 Applications

We conclude by describing three computer vision applications. We describe convolutional networks for image classification where the goal is to identify which of a predetermined set of categories the image belongs to. Then we consider object detection, where the goal is to identify multiple objects in an image and find the bounding box around each. Finally, we describe an early system for semantic segmentation where the goal is to label each pixel according to the object that it belongs to.

10.5.1 Image classification

Much of the pioneering work on deep learning in computer vision was focused on image classification using the ImageNet dataset (figure 10.15). This contains 1,281,167 training images, 50,000 validation images, and 100,000 test images, and every image is labeled as belonging to one of 1000 possible categories.

Most methods reshape the input images to a standard size; in a typical system, the input $\mathbf{x}$ to the network is a 224×224 RGB image and the output is a probability distribution over the 1000 classes. The task is challenging; there are a large number of classes and they exhibit considerable variation (figure 10.15). In 2011, before deep neural networks were applied, the state of the art classified the test images with $\sim 25\%$ errors for the correct class being in the top five suggestions. Five years later, the best machine learning methods eclipsed human performance.

AlexNet (Krizhevsky et al. 2012) was the first convolutional network to perform

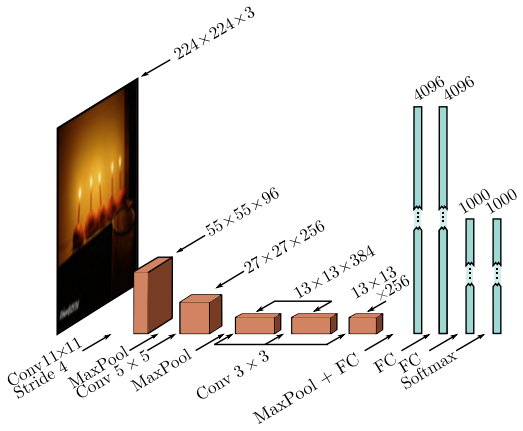

Figure 10.16 AlexNet. The input is a 224×224 color image and the output is a 1000 dimensional vector representing class probabilities. The network first convolves with 11×11 kernels and stride 4 to create 96 channels. It decreases the resolution again using a max-pool operation and applies a 5×5 convolutional layer. Another max-pooling layer follows and three 3×3 convolutional layers are applied. After a final max-pooling operation, the result is vectorized and passed through three fully connected (FC) layers and finally the softmax layer.

well on this task. It consists of eight hidden layers, of which the first five layers are convolutional, and the remaining layers are fully connected (figure 10.16). The network starts by downsampling the input using an 11×11 kernel with a stride of four to create 96 channels. It then downsamples again using a max-pooling layer, before applying a 5×5 kernel to create 256 channels. There are three more convolutional layers with kernel size 3×3, eventually resulting in a 13×13 representation with 256 channels. This is resized into a single vector of length $43,264$ and then passed through three fully connected layers containing 4096, 4096, and 1000 hidden units respectively. The last layer is passed through the softmax function to generate probabilities for the 1000 classes. The full network contains 60 million parameters, most of which are in the fully connected layers.

Problem 10.15

The dataset size was augmented by a factor of 2048 using (i) spatial transformations and (ii) modifications of the input intensities. At test time five different cropped and mirrored versions of the image were run through the network and their predictions averaged. The system was learned using SGD with a momentum coefficient of 0.9 and a batch size of 128. Dropout was applied in the fully connected layers, and a L2 (weight decay) regularizer was used. This system achieved a 16.4% top-5 error rate and a 38.1% top-1 error rate. At the time, this was an enormous leap forward in performance at a task that was considered to be far beyond the capabilities of contemporary methods.

The VGG network (Simonyan & Zisserman 2014) was also targeted at classification in the ImageNet task, and achieved considerably better performance of 6.8% top-5 error rate and a 23.7% top-1 error rate. This network is also composed of a series of interspersed convolutional and max-pooling layers, followed by three fully connected layers (figure 10.17). It was also trained using data augmentation, weight decay, and dropout.

Although there were various minor differences in the training regime, the most important change between AlexNet and VGG was the depth of the network. The latter used 19 hidden layers and 144 million parameters. The networks in figures 10.16 and 10.17 are depicted at the same scale for comparison. There was a general trend for several years for performance on this task to improve as the depth of the networks increased, and this is evidence that depth is important in neural networks.

Problem 10.16

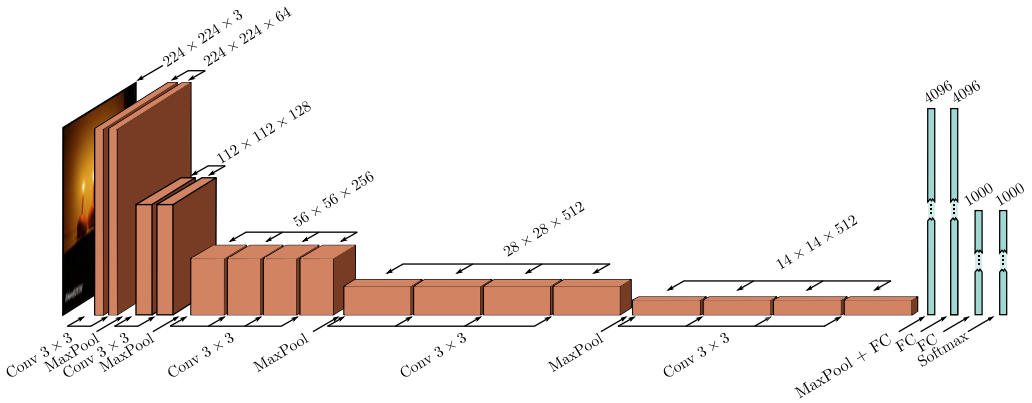

Figure 10.17 The VGG network depicted at the same scale as AlexNet in figure 10.16. This network consists of a series of convolutional layers and max-pooling operations, in which the spatial scale of the representation is gradually decreased but the number of channels gradually increased. The hidden layer after the last convolutional operation is resized to a vector and three fully connected layers follow. The network outputs 1000 activations corresponding to the class labels that are passed through a softmax function to create class probabilities.

10.5.2 Object detection

A second application of convolutional networks is object detection; here the goal is to identify and localize multiple objects within the image. An early deep object detection method was *You Only Look Once* or *YOLO* for short (Redmon et al. 2016). The input to the YOLO network is a 448×448 RGB image. This is passed through 24 convolutional layers that gradually decrease the representation size using max-pooling operations while concurrently increasing the number of channels in a similar way to the VGG network. The final convolutional layer is of size 7×7 and has 1024 channels. This is reshaped to a vector and a fully connected layer maps it to 4096 values. One further fully connected layer maps this representation to the output.

The output values encode which class is present at each of a 7×7 grid of locations (figure 10.18a). For each location, the output values also encode a fixed number of bounding boxes (figure 10.18b). Each box is defined by five parameters: the x- and y-positions of the center, the height and width of the box, and the confidence of the prediction (figure 10.18c). The confidence estimates the overlap between the predicted and ground truth bounding boxes. The system is trained using momentum, weight decay, dropout, and data augmentation. Transfer learning is employed; the network is originally trained on the ImageNet classification task and is then fine-tuned for object detection

After the network is run, a heuristic process is used to remove rectangles with low confidence, and to suppress predicted bounding boxes that correspond to the same object, so that only the most confident one is retained. Example results are shown in figure 10.19.

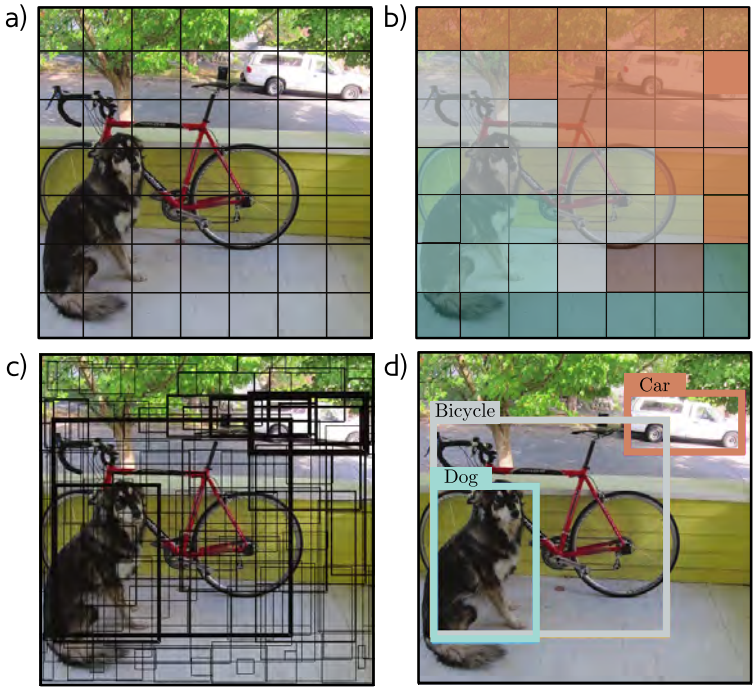

Figure 10.18 YOLO object detection. a) The input image is reshaped to 448×448 and divided into a regular 7×7 grid. b) The system predicts the most likely class at each grid cell. c) It also predicts two bounding boxes per cell, as well as a confidence value (represented by thickness of line). d) During inference, the most likely bounding boxes are retained and boxes with lower confidence values that belong to the same object are suppressed. Adapted from Redmon et al. (2016).

10.5.3 Semantic segmentation

The goal of semantic segmentation is to assign a label to each pixel according to the object that it belongs to, or no label if that pixel does not correspond to anything in the training database. Noh et al. (2015) describe an early network for semantic segmentation (figure 10.20). The input is a 224×224 RGB image and the output is a 224×224×21 array that contains the probability of each of 21 possible classes at each position.

The first part of the network is a smaller version of VGG (figure 10.17) that contains thirteen rather than fifteen convolutional layers and downsizes the representation to size 14×14. There is then one more max-pooling operation, followed by two fully connected layers that map all the hidden units to representations of size 4096. These layers have no spatial position, and so aggregate information from across the whole image.

Here, the architecture diverges from VGG. Another fully connected layer reconstitutes the representation into 7×7 spatial positions and 512 channels. This is followed

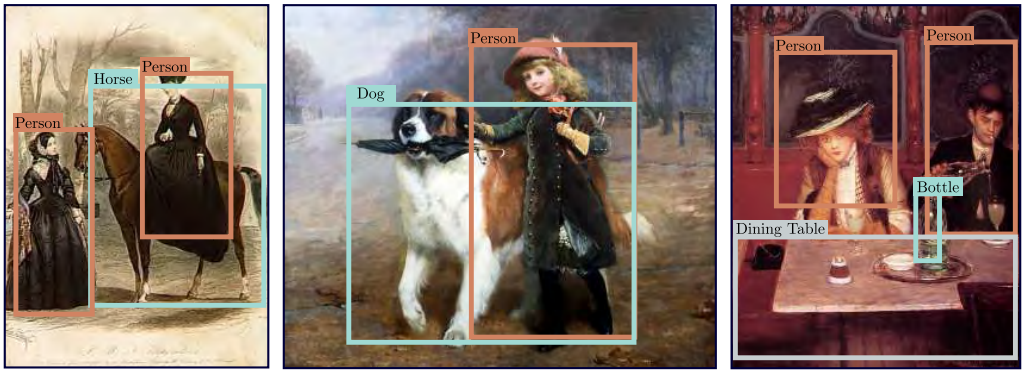

Figure 10.19 YOLO results. The object detection system works well even on paintings that are quite different in style from the training corpus. Adapted from Redmon et al. (2016).

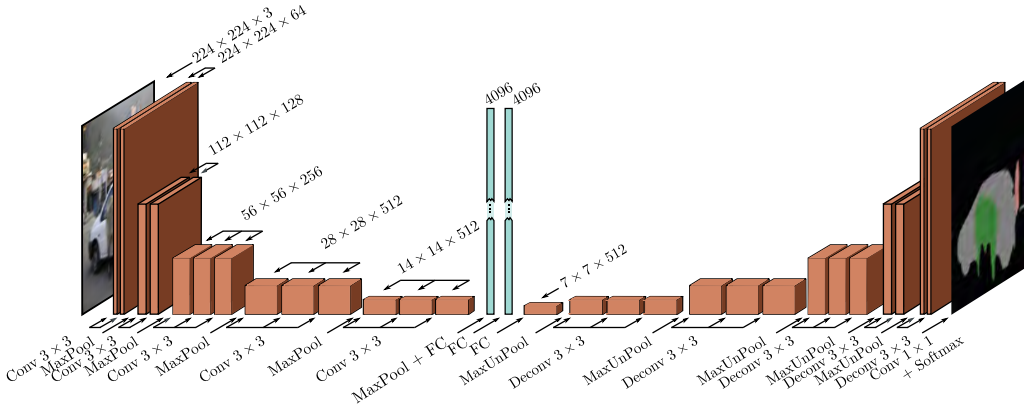

Figure 10.20 Semantic segmentation network of Noh et al. (2015). The input is a 224 × 224 image and this is passed through a version of the VGG network and eventually transformed into a representation of size 4096 using a fully connected layer. This contains information about the entire image. This is then reformed into a representation of size 7×7 using another fully connected layer and the image is upsampled and deconvolved (transposed convolutions without upsampling) in a mirror image of the VGG network. The output is a 224×224×21 representation that gives the output probabilities for the 21 classes at each position.

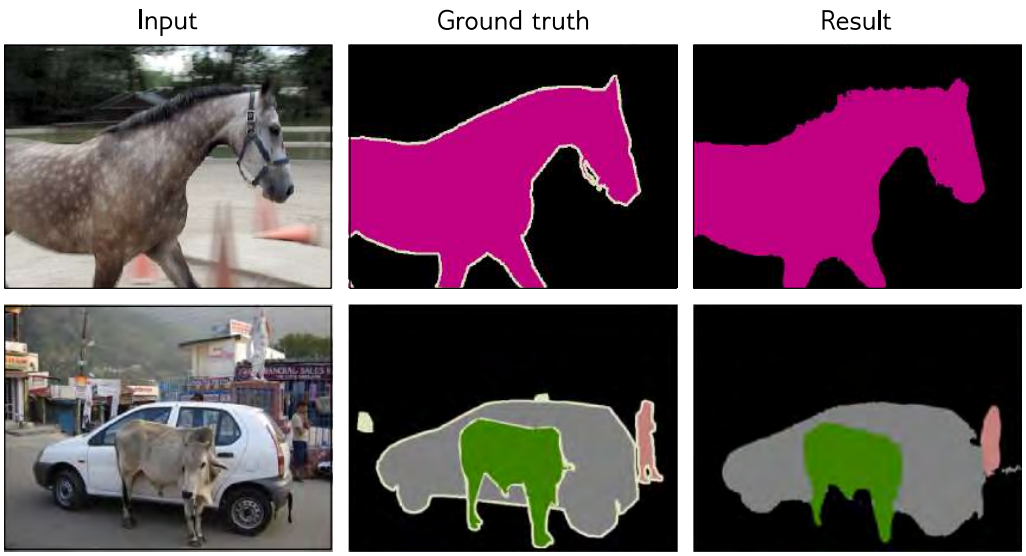

Figure 10.21 Semantic segmentation results. The final result is created from the 21 probability maps by greedily selecting the best class and using a heuristic method to find a sensible binary map based on the probabilities and their spatial proximity. If there is enough evidence, then subsequent classes are added and their segmentation maps are combined. Adapted from Noh et al. (2015).

by a series of max-unpooling layers (see figure 10.12b) and *deconvolution* layers. These are transposed convolutions (see figure 10.13) but in 2D and without the upsampling. Finally, there is a 1×1 convolution to create 21 channels representing the possible classes and a softmax operation at each spatial position to map the activations to class probabilities. The downsampling side of the network is sometimes referred to as an *encoder* and the upsampling side as a *decoder*, and so networks of this type are called *encoder-decoder networks* or *hourglass networks* due to their shape.

The final segmentation is generated using a heuristic method that greedily searches for the class that is most represented and infers its region, taking into account the probabilities, but also encouraging connectedness. The system then adds the next object until it deems that there is insufficient evidence to add more (figure 10.21).

10.6 Summary

In this chapter, we discussed convolutional layers. Each hidden unit is constructed by taking a weighted sum of the nearby inputs, adding a bias, and applying an activation function. The weights and the bias are the same at every spatial position and so there

are far fewer parameters than in a fully connected network and this number does not increase as the input size increases. To ensure that information is not lost, this is repeated multiple times to create many channels at each spatial position.

A typical convolution network consists of a series of convolutional layers interspersed with layers that downsample by a factor of two. As the network progresses, the spatial dimensions decrease and the number of channels increases. At the end of the network, there are one or more fully connected layers that integrate information from across the entire input and create the desired output. If the output is an image, then a mirrored network structure upsamples back to the original size.

For the MNIST1D dataset, the translational equivariance of the convolutional layers imposes a useful inductive bias that increases performance relative to an equivalent fully connected network. We described image classification, object detection, and semantic segmentation networks. Image classification performance was shown to improve as the network became deeper. However, subsequent experimental evidence has shown that increasing the network depth indefinitely doesn't continue to help; after a certain depth, the system becomes difficult to train. This issue was partially resolved by *residual connections*, which are the topic of the next chapter.

Notes

Dumoulin & Visin (2016) present an overview of the mathematics of convolutions that expands on the brief treatment in this chapter.

Convolutional networks: The first multi-layer convolutional networks were developed by LeCun et al. (1989b) and LeCun et al. (1989a) Early applications included handwriting recognition (LeCun et al. 1989a; Martin 1993), face recognition (Lawrence et al. 1997), phoneme recognition (Waibel et al. 1989), spoken word recognition (Bottou et al. 1990), and signature verification (Bromley et al. 1993). However, convolutional networks were popularized by LeCun et al. (1998a), who built a system called LeNet for classifying 28×28 grayscale images of handwritten digits. This is immediately recognizable as a precursor of modern networks; it consisted of a series of convolutional layers, followed by fully connected layers. It used sigmoid activations rather than ReLUs and average-pooling rather than max pooling. AlexNet (Krizhevsky et al. 2012) is widely considered the starting point for modern deep convolutional networks.

ImageNet Challenge: The ImageNet classification challenge drove progress in deep learning for several years after the dramatic improvements of AlexNet. Notable subsequent winners of this challenge include the *network-in-network* architecture (Lin et al. 2013), which alternated convolutions with fully connected layers that operated independently on all of the channels at each position (i.e., 1×1 convolutions). Zeiler & Fergus (2014) and Simonyan & Zisserman (2014) made progress by training larger and deeper architectures that were fundamentally similar to AlexNet. Szegedy et al. (2017) developed an architecture called *GoogLeNet*, which introduced *inception blocks*. These use several parallel paths with different filter sizes, which are then recombined. This effectively allowed the system to learn the filter size.

The trend was for performance to improve with increasing depth; however, it ultimately became difficult to train deeper networks without modifications; these include residual connections

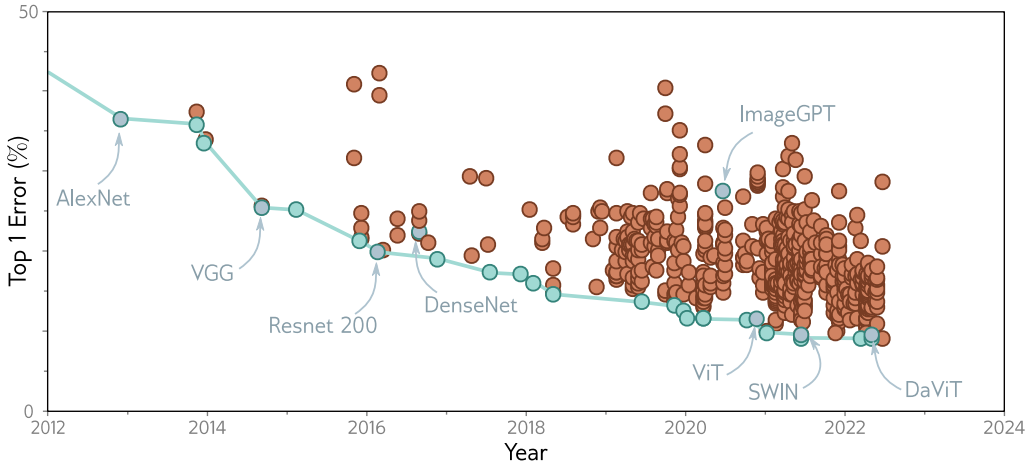

Figure 10.22 ImageNet performance. Each circle represents a different published model. Blue circles represent models that were state-of-the-art. Models discussed in this book are also highlighted. The AlexNet and VGG networks were remarkable for their time, but are now far from state of the art. ResNet 200 and DenseNet are discussed in chapter 11. ImageGPT, ViT, SWIN, and DaViT are discussed in chapter 12. Adapted from https://paperswithcode.com/sota/image-classification-on-imagenet.

and normalization layers, both of which are described in the next chapter. Progress in the ImageNet challenges is summarized in Russakovsky et al. (2015). A more general survey of image classification using convolutional networks can be found in Rawat & Wang (2017). The progress of image classification networks over time is visualized in figure 10.22.

Types of convolutional layer: Atrous or dilated convolutions were introduced by Chen et al. (2014a) and Yu & Koltun (2015). Transposed convolutions were introduced by Long et al. (2015). However, Odena et al. (2016) pointed out that they can lead to checkerboard artifacts and should be used with caution. Convolution with 1×1 filters was introduced Lin et al. (2013).

Many variants of the standard convolutional layer aim to reduce the number of parameters. These include *depthwise* or *channel-separate convolution* (Howard et al. 2017; Tran et al. 2018), in which each channel is separately convolved by a different filter to create a new set of channels. For a kernel size of $K \times K$ with C input channels and C output channels, this requires $K \times K \times C$ parameters rather than the $K \times K \times C \times C$ parameters in a regular convolutional layer. A related approach is *grouped convolutions* (Xie et al. 2017), where each convolution kernel is only applied to a subset of the channels with a commensurate reduction in the parameters. In fact, grouped convolutions were used in AlexNet for computational reasons; the whole network could not run on a single machine and so some channels were processed on one machine and some on another, with only limited points of interaction. *Separable convolutions* treat each kernel as an outer product of 1D vectors, which would use $C + K + K$ parameters for each of the C channels. *Partial convolutions* (Liu et al. 2018a) are used when inpainting missing pixels, and take account of the fact that some of the input is masked. *Gated convolutions* learn the mask

from the previous layer (Yu et al. 2019; Chang et al. 2019).

Downsampling and upsampling: Average pooling dates back to at least LeCun et al. (1989a), and max pooling to Zhou & Chellappa (1988). Scherer et al. (2010) compared these methods and concluded that max pooling was superior. The max unpooling method was introduced by Zeiler et al. (2011) and Zeiler & Fergus (2014). Max pooling can be thought of as applying an ℓ_∞ norm to the hidden units that are to be pooled. This led to applying other norms (Springenberg et al. 2014; Sainath et al. 2013) although these require more computation and are not widely used. Zhang (2019) introduced *max-blur-pooling*, in which a low-pass filter is applied before downsampling to prevent aliasing and showed that this improves generalization.

Shi et al. (2016) introduced *PixelShuffle*, which used convolutional filters with a stride of $1/s$ to scale up 1D signals by a factor of s. To create each output, only the weights that lie exactly on positions are used, and the ones that fall between positions are discarded. This can be implemented by multiplying the number of channels in the kernel by a factor of s, where the s^{th} output position is computed from just the s^{th} subset of channels. This can be trivially extended to 2D convolution, where we now require s^2 channels.

Convolution in 1D and 3D: Convolutional networks are most commonly on images but have also been applied to 1D data in applications that include speech recognition (Abdel-Hamid et al. 2012), sentence classification (Zhang et al. 2015; Conneau et al. 2016), electrocardiogram classification (Kiranyaz et al. 2015), and bearing fault diagnosis (Eren et al. 2019). A survey of the use of 1D convolutional networks can be found in Kiranyaz et al. (2021). Convolutional networks have also been applied to 3D data, including video (Ji et al. 2012; Saha et al. 2016; Tran et al. 2015) and volumetric measurements (Wu et al. 2015b; Maturana & Scherer 2015).

Invariance and equivariance: Part of the motivation for convolutional layers is that they are approximately equivariant with respect to translation, and part of the motivation for max-pooling is to induce invariance to small translations. Zhang (2019) considers the degree to which convolutional networks really have these properties and proposes the max-blur-pooling modification that demonstrably improves them. There is considerable interest in making networks equivariant or invariant to other types of transformation like reflections, rotations, and scaling. Sifre & Mallat (2013) constructed a system based on wavelets that induced both translational and rotational invariance in image patches and applied this to texture classification. Kanazawa et al. (2014) developed locally scale-invariant convolutional neural networks. Cohen & Welling (2016) exploited group theory to construct *group CNNs*, which are equivariant to larger families of transformations including reflections and rotations. Esteves et al. (2017) introduced *polar transformer networks*, which are invariant to translations and equivariant to rotation and scale. Worrall et al. (2017) developed *harmonic networks*, which was the first example of a group CNN that was equivariant to continuous rotations.

Initialization and regularization: Convolutional networks are typically initialized using Xavier initialization (Glorot & Bengio 2010) or He initialization (He et al. 2015) as described in section 7.3 . However, the *ConvolutionOrthogonal* initializer (Xiao et al. 2018) is specialized for convolutional networks (Xiao et al. 2018). Using this initialization, networks of up to 10,000 layers can be trained without residual connections.

Problem 10.17

Dropout is effective for fully connected networks but less so for convolutional layers (Park & Kwak 2016). This may be because neighboring image pixels are highly correlated and so if a hidden unit drops out, the same information is passed on via adjacent positions. This

is the motivation for spatial dropout and cutout. In spatial dropout (Tompson et al. 2015) entire feature maps are discarded instead of individual pixels. This circumvents the problem of neighboring pixels carrying the same information. Similarly, DeVries & Taylor (2017b) propose *cut-out*, in which a square patch of each input image is masked at training time. Wu & Gu (2015) modified max pooling for dropout layers using a method that involves sampling from a probability distribution over the constituent elements rather than always taking the maximum.

Adaptive Kernels The *inception block* (Szegedy et al. 2017) applies convolutional filters of different sizes in parallel and as such provides a crude mechanism by which the network can learn the appropriate filter size. Other work has investigated learning the scale of convolutions as part of the training process (e.g., Pintea et al. 2021; Romero et al. 2021) or the stride of downsampling layers (Riad et al. 2022).

In some systems, the kernel size is changed adaptively based on the data. This is sometimes in the context of guided convolution where one input is used to help guide the computation from another input. For example, an RGB image might be used to help upsample a low-resolution depth map. In Jia et al. (2016) the filter weights themselves are directly predicted by a different network branch. Xiong et al. (2020b) change the kernel size adaptively. Su et al. (2019a) moderate weights of fixed kernels by a function learned from another modality. Dai et al. (2017) learn offsets of weights so that they do not have to be applied in a regular grid.

Visualizing Convolutional Networks The dramatic success of convolutional networks led to a series of efforts to understand exactly what information they extracted from the image by visualizing various aspects of the network, many of which are reviewed in Qin et al. (2018). Erhan et al. (2009) visualized the optimal stimulus that activated a given hidden unit, by starting with an image containing noise, and then optimizing it to make the hidden unit most active using gradient ascent. Zeiler & Fergus (2014) trained a network to reconstruct the input image and then set all the hidden units to zero except the one they were interested in; the reconstruction then provides information about what drives the hidden unit. Mahendran & Vedaldi (2015) attempted to visualize an entire layer of a network. Their *network inversion* technique aimed to find an image that resulted in the activations at that layer, but also applied a generic natural image prior to encourage this image to have sensible properties.

Finally Bau et al. (2017) introduced *network dissection*. Here, a series of images with known pixel labels capturing color, texture, and object type are passed through the network, and the correlation of a hidden unit with each property is measured. This method has the advantage that it only used the forward pass of the network and does not require optimization. These methods did provide some partial insight into how the network processes images. For example, Bau et al. (2017) showed that earlier layers tended to correlate more with texture and color and later layers with the object type. However, it is fair to say that fully understanding the processing of networks containing millions of parameters is currently not possible.

Problems

Problem 10.1 Show that the operation in equation 10.4 is equivariant with respect to translation.

Problem 10.2 Write out the equation for the 1D convolution with a kernel size of three and a stride of two as pictured in figure 10.3a–b.

Problem 10.3 Write out the equation for the 1D dilated convolution with a kernel size of three and a dilation rate of one as pictured in figure 10.4d.

Problem 10.4 Write out the equation for a 1D convolution with kernel size seven, dilation rate of two, and stride of three.

Problem 10.5 Draw weight matrices in the style of figure 10.4d for (i) the strided convolution in figure 10.3a–b, (ii) the convolution with kernel size 5 in figure 10.5c, and (iii) the dilated convolution in figure 10.3d.

Problem 10.6 Draw a 6×12 weight matrix in the style of figure 10.4d relating the inputs $x_1 \ldots x_6$ to the outputs $h_1 \ldots h_{12}$ in the multi-channel convolution as depicted in figures 10.5a–b.

Problem 10.7 Draw a 12×6 weight matrix in the style of figure 10.4d relating the inputs $h_1 \ldots h_{12}$ to the outputs $h'_1 \ldots h'_6$ in the multi-channel convolution in figure 10.5c.

Problem 10.8 The first layers of a convolutional network consists of an input $\mathbf{x}$ and two hidden layers $\mathbf{H}_1$ and $\mathbf{H}_2$. The input has one channel. The first hidden layer is computed using a kernel size of three and has four channels. The second hidden layer is computed using a kernel size of five and has ten channels. How many biases and how many weights are needed for each of these two convolutional layers?

Problem 10.9 Consider a convolutional network with 1D input $\mathbf{x}$. The first hidden layer $\mathbf{H}_1$ is computed using a convolution with kernel size five, stride two, and with a dilation rate of zero. The second hidden layer $\mathbf{H}_2$ is computed using a convolution with kernel size three, stride one, and with a dilation rate of zero. The third hidden layer $\mathbf{H}_3$ is computed using a convolution with kernel size three, stride one, and a dilation rate of one. What are the receptive field sizes at each hidden layer?

Problem 10.10 The 1D convolutional network in figure 10.7 was trained using stochastic gradient descent with a learning rate of 0.01 and a batch size of 100 on a training dataset of 4,000 examples for 100,000 steps. How many epochs was the network trained for?

Problem 10.11 Draw a weight matrix in the style of figure 10.4d that shows the relationship between the 24 inputs and the 24 outputs in figure 10.9.

Problem 10.12 Draw a weight matrix in the style of figure 10.4d that samples every other variable in a 1D input (i.e., the 1D analog of figure 10.11a). Show that the weight matrix for 1D convolution with kernel size and stride two is equivalent to composing the matrices for 1D convolution with kernel size one and this sampling matrix.

Problem 10.13 Write Python code to take a grayscale image, and downsample it three times using (i) subsampling, (ii) mean pooling, and (iii) max pooling.

Problem 10.14 Write Python to re-upsample the image from problem 10.14 using (i) duplication, (ii) max-unpooling, and (iii) bilinear interpolation.

Problem 10.15 What is the receptive field size at each layer of AlexNet (figure 10.16)?

Problem 10.16 How many weights and biases are there at each convolutional layer and fully connected layer in the VGG architecture (figure 10.17)?

Problem 10.17 Consider two hidden layers of size 224×224 with C_1 and C_2 channels respectively that are connected by a 3×3 convolutional layer. Describe how to initialize the weights using He initialization.

Chapter 11

Residual networks

The previous chapter described how image classification performance improved as the depth of convolutional networks was extended from eight layers (AlexNet) to eighteen layers (VGG). This led to experimentation with even deeper networks. However, performance decreased again when many more layers were added.

This chapter introduces *residual blocks*. Here, each network layer computes an additive change to the current representation instead of transforming it directly. This allows deeper networks to be trained but causes an exponential increase in the activation magnitudes at initialization. To compensate for this, residual blocks employ *batch normalization*, which resets the mean and variance of the activations at each layer.

Residual blocks with batch normalization allow much deeper networks to be trained, and these networks improve performance across a variety of tasks. Architectures that combine residual blocks to tackle image classification, medical image segmentation, and human pose estimation are described.

11.1 Sequential processing

Every network we have seen so far processes the data sequentially; each layer receives the output of the previous layer and passes the result to the next (figure 11.1). For example, a three-layer network is defined by:

$$\begin{aligned}\mathbf{h}_1 &= \mathbf{f}_1[\mathbf{x}, \boldsymbol{\phi}_1] \\ \mathbf{h}_2 &= \mathbf{f}_2[\mathbf{h}_1, \boldsymbol{\phi}_2] \\ \mathbf{h}_3 &= \mathbf{f}_3[\mathbf{h}_2, \boldsymbol{\phi}_3] \\ \mathbf{y} &= \mathbf{f}_4[\mathbf{h}_3, \boldsymbol{\phi}_4],\end{aligned} \tag{11.1}$$

where $\mathbf{h}_1$, $\mathbf{h}_2$, and $\mathbf{h}_3$ denote the intermediate hidden layers, $\mathbf{x}$ is the network input, $\mathbf{y}$ is the output, and the functions $\mathbf{f}_k[\bullet, \boldsymbol{\phi}_k]$ perform the processing.

In a standard neural network, each layer consists of a linear transformation followed by an activation function, and the parameters $\boldsymbol{\phi}_k$ comprise the weights and biases of the

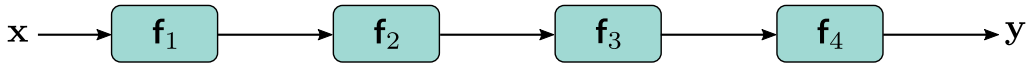

Figure 11.1 Sequential processing. Standard neural networks pass the output of each layer directly into the next layer.

linear transformation. In a convolutional network, each layer consists of a set of convolutions followed by an activation function, and the parameters comprise the convolutional kernels and biases.

Since the processing is sequential, we can equivalently think of this network as a series of nested functions:

$$\mathbf{y} = \mathbf{f}_4\Big[\mathbf{f}_3\Big[\mathbf{f}_2\big[\mathbf{f}_1[\mathbf{x}, \boldsymbol{\phi}_1], \boldsymbol{\phi}_2\big], \boldsymbol{\phi}_3\Big], \boldsymbol{\phi}_4\Big]. \tag{11.2}$$

11.1.1 Limitations of sequential processing

In principle, we can add as many layers as we want, and in the previous chapter we saw that adding more layers to a convolutional network does improve performance; the VGG network (figure 10.17), which has eighteen layers, outperforms AlexNet (figure 10.16), which has eight layers. However, as further layers are added, image classification performance decreases again (figure 11.2). This happens for the training set as well as the test set, which implies that the problem is training deeper networks, rather than the inability of deeper networks to generalize.

This phenomenon is not completely understood but one conjecture is that at initialization, the loss gradients change unpredictably when we modify parameters in an early network layer. With appropriate initialization, the gradient of the loss with respect to these parameters will be reasonable (i.e., no exploding or vanishing gradients). However, the derivative assumes an infinitesimal change in the parameter whereas optimization algorithms change the parameter by a finite step size. It may be that any reasonable choice of step size moves to a place with a completely different and unrelated gradient; the loss surface looks like an enormous range of tiny mountains, rather than a single large-scale structure that is easy to descend. Consequently, the algorithm does not make progress in the way that it does when the loss function gradient changes more slowly.

This conjecture is supported by empirical observations of the gradients in networks with a single input and a single output. For a shallow network, the gradient of the output with respect to the input changes relatively slowly as we change the input (figure 11.3a). However, for a deep network with 24 layers, a tiny change in the input results in a completely different gradient (figure 11.3b). This is captured by the autocorrelation function of the gradient (figure 11.3c). For shallow networks, the gradients stay highly correlated as the input changes, but for deep networks, their correlation quickly drops to zero. This is termed the *shattered gradients* phenomenon.

Appendix C.3 autocorrelation function

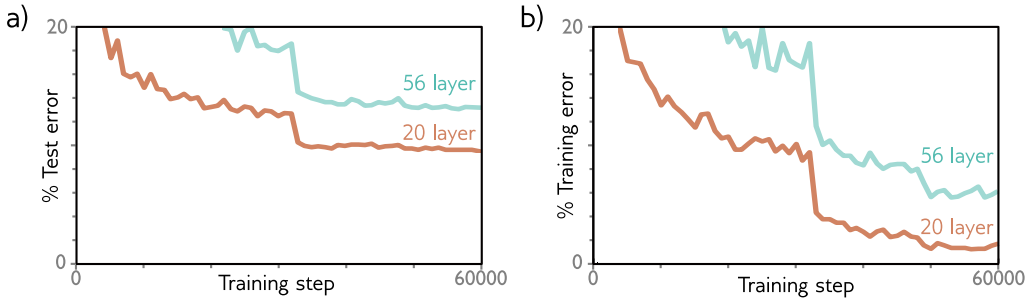

Figure 11.2 Decrease in performance when adding more convolutional layers. a) A 20-layer convolutional network outperforms a 56-layer neural network for image classification on the test set of the CIFAR-10 dataset (Krizhevsky & Hinton 2009). b) This is also true for the training set, which suggests that the problem relates to training the original network, rather than a failure to generalize to new data. Adapted from He et al. (2016a).

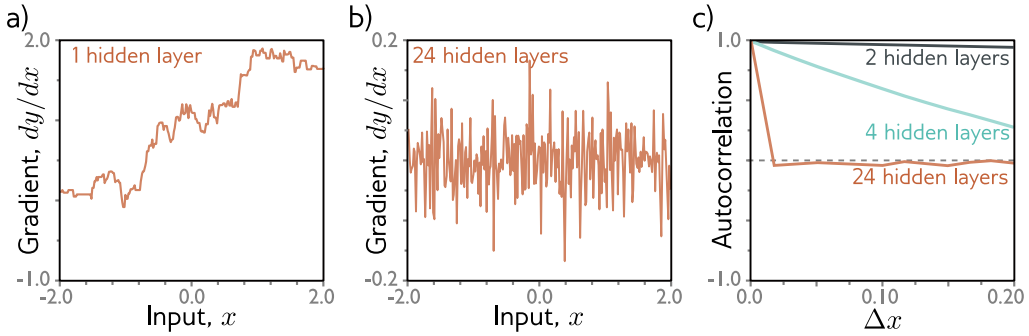

Figure 11.3 Shattered gradients. a) Consider a shallow network with 200 hidden units and Glorot initialization (He initialization without the factor of two) for both the weights and biases. The gradient $\partial y/\partial x$ of the scalar network output y with respect to the scalar input x changes relatively slowly as we change the input x. b) For a deep network with 24 layers and 200 hidden units per layer, this gradient changes very quickly and unpredictably. c) The autocorrelation function of the gradient shows that nearby gradients become unrelated (have autocorrelation close to zero) for deep networks. This *shattered gradients* phenomenon may explain why it is hard to train deep networks. Gradient descent algorithms rely on the loss surface being relatively smooth, so the gradients should be related before and after each update step. Adapted from Balduzzi et al. (2017).

Shattered gradients presumably arise because changes in early network layers modify the output in an increasingly complex way as the network becomes deeper. The derivative of the output $\mathbf{y}$ with respect to the first layer $\mathbf{f}_1$ of the network in equation 11.1 is:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{f}_1} = \frac{\partial \mathbf{f}_4}{\partial \mathbf{f}_3}\frac{\partial \mathbf{f}_3}{\partial \mathbf{f}_2}\frac{\partial \mathbf{f}_2}{\partial \mathbf{f}_1}. \tag{11.3}$$

When we change the parameters that determine $\mathbf{f}_1$, *all* of the derivatives in this sequence can change, since layers $\mathbf{f}_2, \mathbf{f}_3$ and $\mathbf{f}_4$ are themselves computed from $\mathbf{f}_1$. Consequently, the updated gradient at each training example may be completely different, and the loss function becomes badly behaved.[1]

11.2 Residual connections and residual blocks

Residual or *skip connections* are branches in the computational path, whereby the input to each network layer $\mathbf{f}[\bullet]$ is added back to the output (figure 11.4a). By analogy to equation 11.1, the residual network is defined as:

$$\begin{aligned}
\mathbf{h}_1 &= \mathbf{x} + \mathbf{f}_1[\mathbf{x}, \boldsymbol{\phi}_1] \\
\mathbf{h}_2 &= \mathbf{h}_1 + \mathbf{f}_2[\mathbf{h}_1, \boldsymbol{\phi}_2] \\
\mathbf{h}_3 &= \mathbf{h}_2 + \mathbf{f}_3[\mathbf{h}_2, \boldsymbol{\phi}_3] \\
\mathbf{y} &= \mathbf{h}_3 + \mathbf{f}_4[\mathbf{h}_3, \boldsymbol{\phi}_4],
\end{aligned} \tag{11.4}$$

where the first term on the right-hand side of each line corresponds to the residual connection. Each function $\mathbf{f}_k$ learns an additive change to the current representation. It follows that their outputs must be the same size as their inputs. Each additive combination of the input and the processed output is known as a *residual block*.

Problem 11.1

Once more, we can write this as a single function by substituting in the expressions for the intermediate quantities $\mathbf{h}_k$:

$$\begin{aligned}
\mathbf{y} = \mathbf{x} &+ \mathbf{f}_1[\mathbf{x}] \\
&+ \mathbf{f}_2\big[\mathbf{x} + \mathbf{f}_1[\mathbf{x}]\big] \\
&+ \mathbf{f}_3\Big[\mathbf{x} + \mathbf{f}_1[\mathbf{x}] + \mathbf{f}_2\big[\mathbf{x} + \mathbf{f}_1[\mathbf{x}]\big]\Big] \\
&+ \mathbf{f}_4\Big[\mathbf{x} + \mathbf{f}_1[\mathbf{x}] + \mathbf{f}_2\big[\mathbf{x} + \mathbf{f}_1[\mathbf{x}]\big] + \mathbf{f}_3\Big[\mathbf{x} + \mathbf{f}_1[\mathbf{x}] + \mathbf{f}_2\big[\mathbf{x} + \mathbf{f}_1[\mathbf{x}]\big]\Big]\Big],
\end{aligned} \tag{11.5}$$

where we have omitted the parameters $\boldsymbol{\phi}_\bullet$ for clarity. We can think of this equation as "unraveling" the network (figure 11.4b). We see that the final network output is a sum of the input and four smaller networks, corresponding to each line of the equation; one

[1] In equations 11.3 and 11.6 we overload notation to define $\mathbf{f}_k$ as the output of the function $\mathbf{f}_k[\bullet]$.

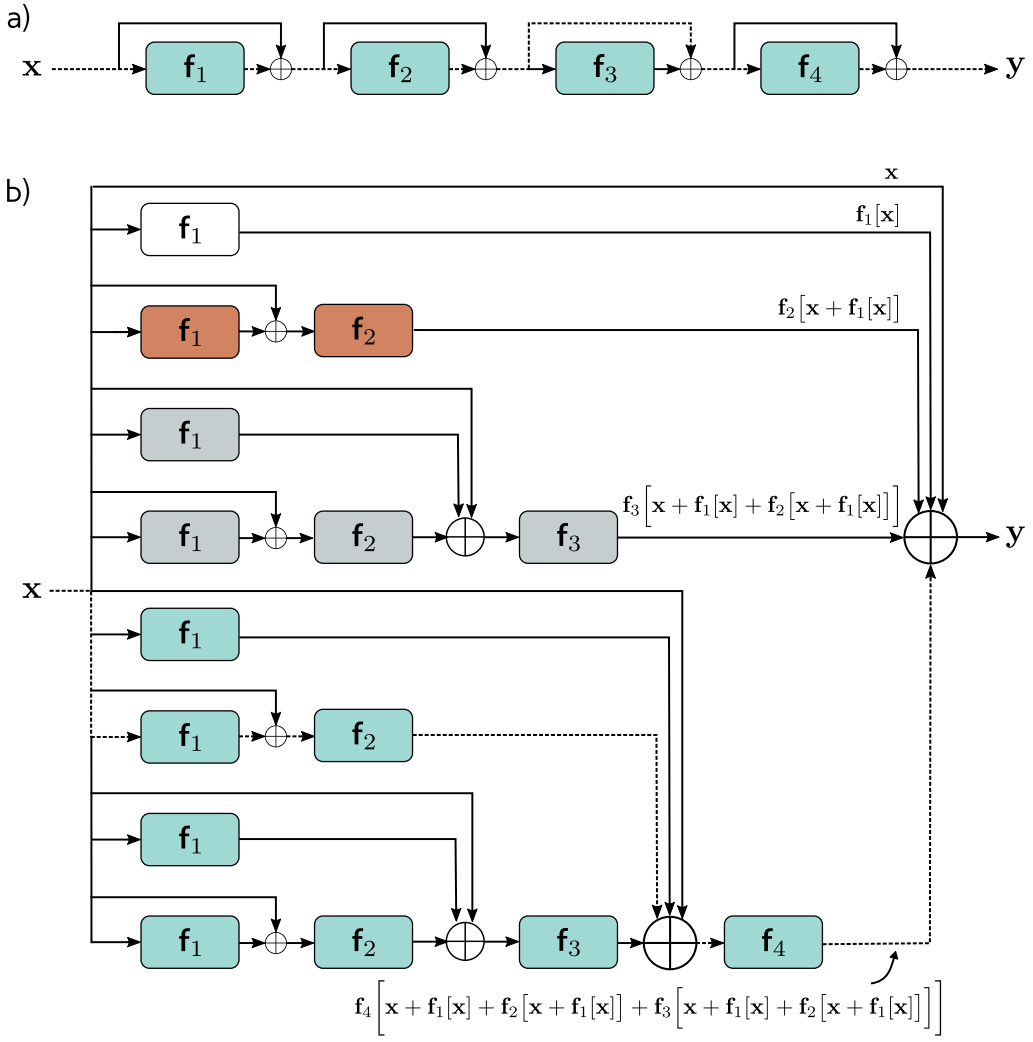

Figure 11.4 Residual connections. a) The output of each function $\mathbf{f}_k[\mathbf{x}, \boldsymbol{\phi}_k]$ is added back to its input, which is passed via a parallel computational path called a residual or skip connection. Hence, the function computes an additive change to the representation. b) Upon expanding (unraveling) the network equations, we find that the output is the sum of the input plus four smaller networks (depicted in white, orange, gray, and cyan, respectively, and corresponding to terms in equation 11.5); we can think of this as an ensemble of networks. Moreover, the output from the cyan network is itself a transformation $\mathbf{f}_4[\bullet, \boldsymbol{\phi}_4]$ of another ensemble and so on. Alternatively, we can consider the network as a combination of 16 different paths through the computational graph. One example is the dashed path from input $\mathbf{x}$ to output $\mathbf{y}$ which is the same in both panels (a) and (b).

interpretation is that residual connections turn the original network into an ensemble of these smaller networks whose outputs are summed to compute the result.

A complementary way of thinking about this residual network is that it creates sixteen paths of different lengths from input to output. For example, the first function $\mathbf{f}_1[\mathbf{x}]$ occurs in eight of these sixteen paths, including as a direct additive term, (i.e., a path length of one) and the analogous derivative to equation 11.3 is:

Problem 11.2

Problem 11.3

$$\frac{\partial \mathbf{y}}{\partial \mathbf{f}_1} = \mathbf{I} + \frac{\partial \mathbf{f}_2}{\partial \mathbf{f}_1} + \left(\frac{\partial \mathbf{f}_3}{\partial \mathbf{f}_1} + \frac{\partial \mathbf{f}_3}{\partial \mathbf{f}_2}\frac{\partial \mathbf{f}_2}{\partial \mathbf{f}_1}\right) + \left(\frac{\partial \mathbf{f}_4}{\partial \mathbf{f}_1} + \frac{\partial \mathbf{f}_4}{\partial \mathbf{f}_2}\frac{\partial \mathbf{f}_2}{\partial \mathbf{f}_1} + \frac{\partial \mathbf{f}_4}{\partial \mathbf{f}_3}\frac{\partial \mathbf{f}_3}{\partial \mathbf{f}_1} + \frac{\partial \mathbf{f}_4}{\partial \mathbf{f}_3}\frac{\partial \mathbf{f}_3}{\partial \mathbf{f}_2}\frac{\partial \mathbf{f}_2}{\partial \mathbf{f}_1}\right), \tag{11.6}$$

where there is one term for each of the eight paths. The identity term on the right-hand side shows that changes in the parameters $\boldsymbol{\phi}_1$ in the first layer $\mathbf{f}_1[\mathbf{x}, \boldsymbol{\phi}_1]$ contribute directly to changes in the network output $\mathbf{y}$ as well as indirectly through other chains of derivatives. In general, gradients through the shorter paths will be better behaved.

Consequently, networks with residual links are less likely to suffer from problems due to long chains of derivatives. In addition, if a function $\mathbf{f}_k[\bullet]$ has a very small output (as might be encouraged by an L2 regularization term), then the original signal is largely unmodified. It follows that it is much easier to learn an identity mapping, and hence for the network to choose the effective depth during the training process.

11.2.1 Order of operations in residual blocks

Until this point, we have implied that the additive functions $\mathbf{f}[\mathbf{x}]$ could be any valid network layer (e.g., fully connected, or convolutional). This is technically true, but the order of operations in these additive functions is important. They must contain a nonlinear activation function like a ReLU or the entire network will be linear. However, in a typical network layer (figure 11.5a), the ReLU function is at the end, and so the output is non-negative. If we adopt this convention, then each residual block we will only be able to increase the input values.

Consequently, it is typical to change the order of operations so that the activation function is applied first and is followed by the linear transformation (figure 11.5b). Sometimes there may be several layers of processing within the residual block (figure 11.5c) but these usually terminate with a linear transformation. Finally, we note that if we start each of these blocks with a ReLU operation, then they will do nothing if the initial network input is negative since the ReLU will clip the entire signal to zero. Hence, it's typical to start the network with a linear transformation.

11.2.2 Deeper networks with residual connections

As a rule of thumb, adding residual connections roughly doubles the depth of a network that can be practically trained before performance starts to degrade. However, we would like to be able to increase the depth further. To understand why residual connections

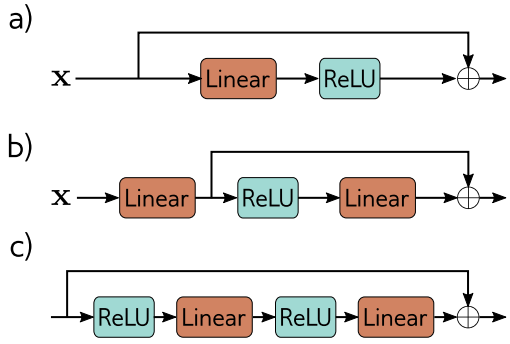

Figure 11.5 Order of operations in residual blocks. a) The usual order of linear transformation or convolution followed by a ReLU nonlinearity means that each residual block can only add non-negative quantities. b) With the reverse order, both positive and negative quantities can be added. However, we must add a linear transformation at the start of the network in case the input is all negative. c) In practice, it's common for a residual block to contain several network layers.

do not allow us to increase the depth arbitrarily, we must consider how the variance of the activations changes during the forward pass, and how the magnitude of the gradients changes during the backward pass.

11.3 Exploding gradients in residual networks

In section 7.3 we saw that initializing the network parameters is critical. Without careful initialization, the magnitudes of the intermediate values during the forward pass of backpropagation can increase or decrease exponentially. Similarly, the gradients during the backward pass can explode or vanish as we move backward through the network.

The standard approach is to initialize the network parameters so that the expected variance of the activations (in the forward pass) and gradients (in the backward pass) remains the same between layers. He initialization (section 7.3) achieves this for ReLU activations by initializing the biases $\boldsymbol{\beta}$ to zero and choosing normally distributed weights $\boldsymbol{\Omega}$ with mean zero and variance $2/D_h$ where D_h is the number of hidden units in the previous hidden layer (see figure 7.4).

Now consider a residual network. We do not have to worry about the intermediate values or gradients vanishing with network depth since there exists a path whereby each layer directly contributes to the network output (equation 11.5 and figure 11.4b). However, even if we use He initialization within the residual block, the values in the forward pass increase exponentially as we move through the network.

To see why, consider that we add the result of the processing in the residual block back to the input. Each of the two branches has some (independent) variability. Hence, the overall variance increases when we recombine them. With ReLU activations and He initialization, the expected variance is unchanged by the processing in each block. Consequently, when we recombine with the input, the variance doubles (figure 11.6a), and so grows exponentially with the number of residual blocks. This limits the possible network depth before floating point precision is exceeded in the forward pass. A similar argument applies to the gradients in the backward pass of the backpropagation algorithm.

Problem 11.4

Problem 11.5

Hence, residual networks still suffer from unstable forward propagation and exploding gradients even with He initialization. One approach that would stabilize the forward and

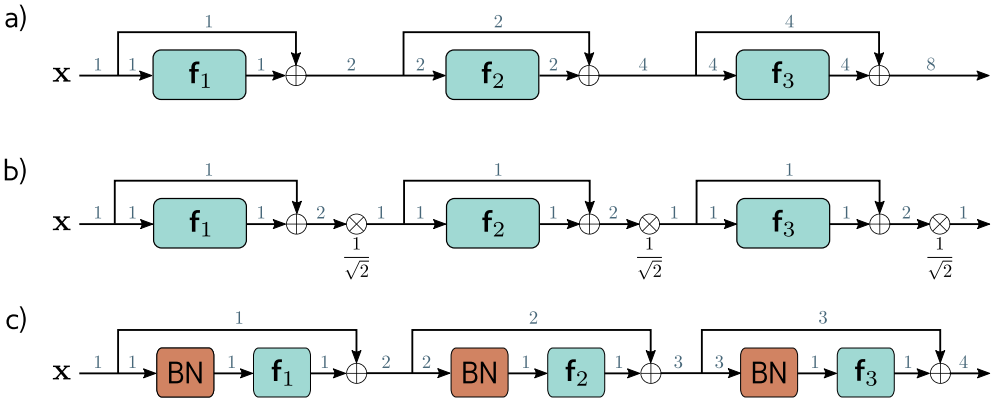

Figure 11.6 Variance in residual networks. a) He initialization ensures that the expected variance remains the same after a linear plus ReLU layer $\mathbf{f}_k$. Unfortunately, in residual networks, the output of each block is added back to the input, and so the variance doubles at each layer (blue numbers indicate variance) and grows exponentially. b) One approach would be to rescale the signal by $1/\sqrt{2}$ between each residual block. c) A second method is to use batch normalization (BN) as the first step in the residual block and set the associated offset δ to zero and scale γ to one. This transforms the input to each layer to have variance one, and with He initialization, the output variance will also be one. Now the variance increases linearly with the number of residual blocks. A side-effect is that at initialization, later layers of the network make a relatively smaller contribution.

Problem 11.6

backward passes would be to use He initialization and then multiply the combined output of each residual block by $1/\sqrt{2}$ to compensate for the doubling (figure 11.6b). However, it is more usual to use *batch normalization*.

11.4 Batch normalization

Batch normalization or *BatchNorm* shifts and rescales each activation h so that its mean and variance across the batch $\mathcal{B}$ become values which are learned during training. First, the empirical mean m_h and standard deviation s_h are computed:

$$\begin{aligned} m_h &= \frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} h_i \\ s_h &= \sqrt{\frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}}(h_i - m_h)^2}. \end{aligned} \tag{11.7}$$

Appendix B.2.3 standardization

Then we use these quantities to normalize the batch activations to have mean zero and

variance one:

$$h_i \leftarrow \frac{h_i - m_h}{s_h + \epsilon} \qquad \forall i \in \mathcal{B}, \tag{11.8}$$

where ϵ is a small number that is present that prevents division by zero if h_i is the same for every member of the batch.

Finally, the normalized variable is scaled by γ and shifted by δ:

$$h_i \leftarrow \gamma h_i + \delta \qquad \forall i \in \mathcal{B}, \tag{11.9}$$

so that after this operation, the activations have mean δ and standard deviation γ across all of the members of the batch. Both of these quantities are parameters of the model and are learned during training. Problem 11.7

Batch normalization is applied independently to each hidden unit, so in a standard neural network with K layers each containing D hidden units, there would be KD learned offsets δ and KD learned scales γ. In a convolutional network, the normalizing statistics are computed over both the batch and the spatial position, so if there were K layers each containing C channels, there would be KC learned offsets and scales. At test time, we do not have a batch from which we can gather statistics. To resolve this, the statistics m_h and σ_h are calculated across the whole training dataset (rather than just a batch) and frozen in the final network. Problem 11.8

11.4.1 Costs and benefits of batch normalization

Batch normalization makes the network invariant to rescaling the weights and biases that contribute to each activation; if these are doubled, then the activations also double, the estimated standard deviation s_h doubles, and the normalization in equation 11.8 compensates for these changes. This happens separately for each hidden unit, and so in practice, there will be a large family of weights and biases that all produce the same effect. Batch normalization also adds two parameters γ and δ at every hidden unit which makes the model somewhat larger. Hence, it both creates redundancy in the weight parameters and adds extra parameters to compensate for that redundancy. This is obviously inefficient, but batch normalization also provides several benefits:

Stable forward propagation: If we initialize the offsets δ to zero and the scales γ to one, then each output activation will have variance one. In a regular network, this ensures that the variance is stable during forward propagation at initialization. In a residual network, the variance must still increase as we are adding a new source of variation to the input at each layer. However, it will increase linearly with each residual block; the k^{th} layer adds one unit of variance to the existing variance of k (figure 11.6c).

This has the side-effect that later layers make a smaller proportional change to the overall variation than earlier ones at initialization. Hence, the network is effectively less deep at the start of training, since later layers make negligible changes. As training proceeds, the network can increase the scales γ in the later layers, and so the network can easily control its own effective depth.

Higher learning rates: Empirical studies and theory both show that batch normalization makes the loss surface and its gradient change more smoothly (i.e., reduces shattered gradients). This means that we can use higher learning rates as the surface is more predictable. We saw in section 9.2 that higher learning rates can improve test performance.

Regularization: We also saw in chapter 9 that adding noise to the training process can make models generalize better. Batch normalization injects noise because the normalization depends on the statistics of the entire batch. Consequently, for a given training example, the activations will be normalized by an amount that depends on the other members of the batch, and so will be processed slightly differently depending on the batch elements.

11.5 Common residual architectures

Residual connections are now a standard part of most deep learning pipelines. This section reviews some well-known architectures that incorporate them.

11.5.1 ResNet

Residual blocks were first used in convolutional networks for image classification. The resulting networks are known as residual networks or *ResNets* for short. In ResNets each residual block contains a batch normalization operation, then the ReLU activation function, and a convolutional layer. This is followed by the same sequence again before being added back to the input (figure 11.7a). Trial and error have shown that this order of operations works well for image classification.

Problem 11.9

For very deep networks, the number of parameters may become undesirably large. *Bottleneck residual blocks* make more efficient use of parameters using three convolutions. The first has a 1×1 kernel and reduces the number of channels. The second is a regular 3×3 kernel and the third is another 1×1 kernel to increase the number of channels back to the original amount (figure 11.7b). In this way, we can integrate information over a 3×3 pixel area using fewer parameters.

Problem 11.10

The ResNet-200 model (figure 11.8) contains 200 layers and is intended for image classification on the ImageNet database (figure 10.15). The architecture is similar to AlexNet and VGG but it uses bottleneck residual blocks instead of vanilla convolutional layers. As with AlexNet and VGG, these are periodically interspersed with decreases in spatial resolution and simultaneous increases in the number of channels. Here, the resolution is decreased by downsampling using convolutions with stride two, and the number of channels is increased either by appending zeros to the representation or using an extra 1×1 convolution. At the start of the network is a 7×7 convolutional layer, followed by a downsampling operation. At the end, there is a fully connected layer that maps the block to a vector of length 1000. This is then passed through a softmax layer to generate the class probabilities.

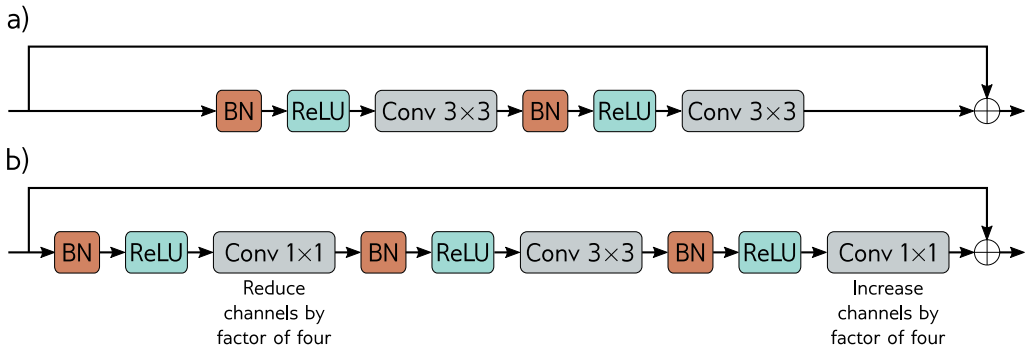

Figure 11.7 ResNet blocks. a) A standard block in the ResNet architecture contains a batch normalization operation, followed by an activation function, and then a 3×3 convolutional layer. Then, this sequence is repeated. b). A bottleneck ResNet block still integrates information over a 3×3 region but uses fewer parameters. It contains three convolutions. The first 1×1 convolution reduces the number of channels. The second 3×3 convolution is applied to the smaller representation. A final 1×1 convolution increases the number of channels again so that it can be added back to the input.

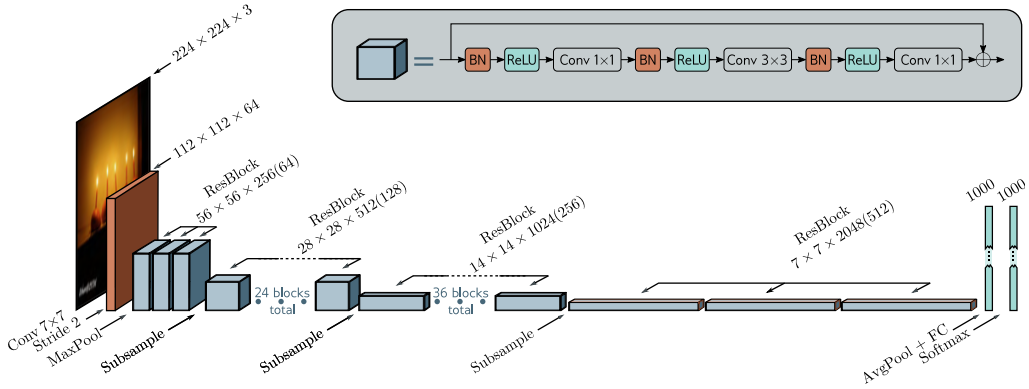

Figure 11.8 ResNet 200 model. A standard 7×7 convolutional layer with stride two is applied, followed by a MaxPool operation. A series of bottleneck residual blocks follow (number in brackets is channels after first 1×1 convolution), with periodic downsampling, and accompanying increases in the number of channels. The network concludes with average pooling across all spatial positions and a fully connected layer that maps to pre-softmax activations.

The ResNet-200 model achieved a remarkable 4.8% error rate for the correct class being in the top five and 20.1% for identifying the correct class correctly. This compares favorably with AlexNet (16.4%, 38.1%) and VGG (6.8%, 23.7%) and was one of the first networks to exceed human performance (5.1% for being in the top five guesses). However, this model was conceived in 2016 and is now far from state-of-the-art. At the time of writing, the best-performing model on this task has a 9.0% error for identifying the class correctly (see figure 10.22). This and all the other top-performing models for image classification are now based on transformers (see chapter 12).

11.5.2 DenseNet

Residual blocks receive the output from the previous layer, modify it, and add it back to the original input. An alternative is to concatenate the modified and original signals. This increases the representation size (in terms of channels for a convolutional network), but an optional subsequent linear transformation can map back to the original size (a 1×1 convolution for a convolutional network). This allows the model to add the representations together, take a weighted sum, or combine them in a more complex way.

The DenseNet architecture uses concatenation so that the input to a layer comprises the concatenated outputs from *all* previous layers (figure 11.9). These are processed to create a new representation that is itself concatenated with the previous representation and passed to the next layer. This concatenation means that there is a direct contribution from earlier layers to the output, and so the loss surface behaves reasonably.

Problem 11.11

In practice, this can only be sustained for a few layers because the number of channels (and hence the number of parameters required to process them) becomes increasingly large. This problem can be alleviated by applying a 1×1 convolution to reduce the number of channels before the next 3×3 convolution is applied. In a convolutional network, the input is periodically downsampled; concatenation of the representation across the downsampling makes no sense since the representations are of different sizes. Consequently, the chain of concatenation is broken at this point and a smaller representation starts a new chain. In addition, another bottleneck 1×1 convolution can be applied when the downsampling occurs to further control the representation size.

This network performs competitively with ResNet models on image classification benchmarks (see figure 10.22); indeed for a comparable parameter count, the DenseNet model can perform better. This is presumably because it has the option of reusing processing from earlier layers more flexibly.

11.5.3 U-Nets and hourglass networks

Section 10.5.3 described a network for semantic segmentation that had an encoder-decoder or hourglass structure. The encoder repeatedly downsamples the image until the receptive fields cover large areas and information is integrated from across the image. Then the decoder upsamples it again until it is the same size as the original image. The final output is a probability over possible object classes at each pixel. One drawback

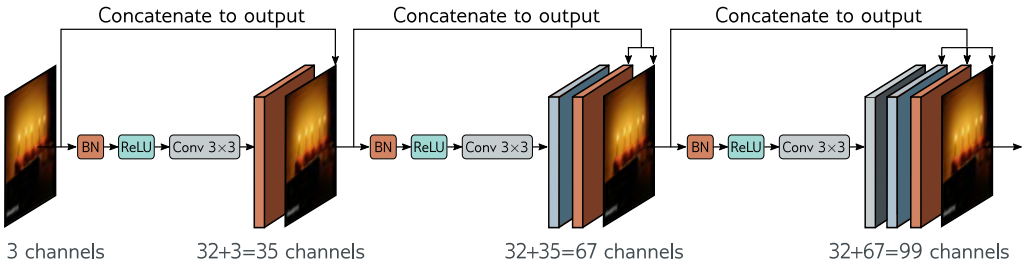

Figure 11.9 DenseNet. This architecture uses residual connections to concatenate the outputs of earlier layers to later ones. Here, the three-channel input image is processed to form a 32-channel representation. The input image is concatenated to this to give a total of 35 channels. This combined representation is processed to create another 32-channel representation and both earlier representations are concatenated to this to create a total of 67 channels and so on.

of this architecture is that the low-resolution representation in the middle of the network has to remember the high-resolution details of the original image to make the final result accurate. This is not necessary if residual connections add or concatenate the representations from the encoder to their partner in the decoder.

The *U-Net* is an encoder-decoder architecture, in which the earlier representations are concatenated to the later ones. Figure 11.10 shows the U-Net applied to segmenting HeLa cells on glass recorded with differential inference contrast. The network uses "valid" convolutions, so the spatial size of the representation decreases by two pixels each time a 3×3 convolutional layer is applied. This means that the upsampled version is smaller than its counterpart in the encoder, which must be cropped before concatenation.

The U-Net has been used extensively for segmenting medical images (figure 11.11), but this and similar architectures have also found use in computer vision. *Hourglass networks* are similar, but apply further convolutional layers in the skip connections and add the result back to the decoder side rather than concatenating it. Several of these networks can be stacked and trained together, creating a *stacked hourglass network* that alternates between considering the image at local and global levels. This network has been used for human pose estimation (figure 11.12). The system is trained to predict one "heatmap" for each joint, and the final joint position is estimated by taking the maximum of each heatmap.

11.6 Why do nets with residual connections perform so well?

Residual networks allow much deeper networks to be trained; in fact, it's possible to extend the ResNet architecture to 1000 layers and still train effectively. The improvement in image classification performance was initially attributed to the additional depth of the

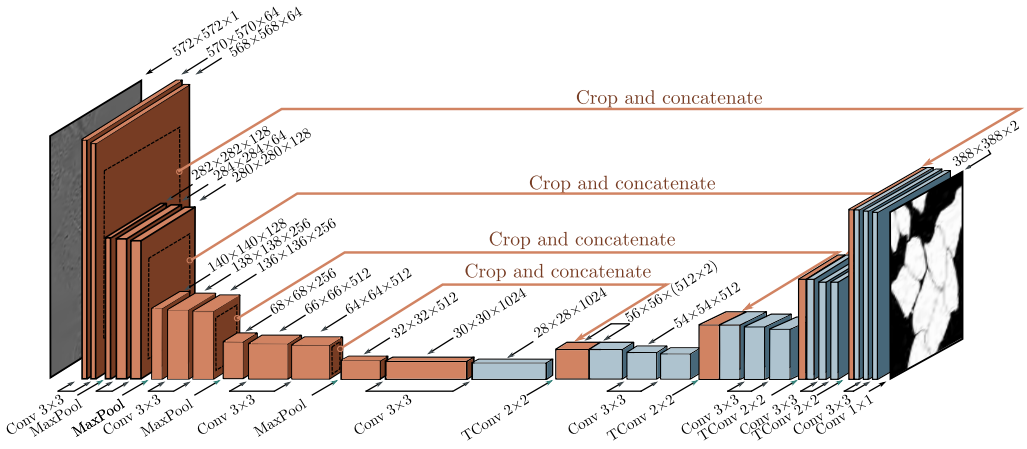

Figure 11.10 U-Net. The network follows an encoder-decoder structure, in which the representation is downsampled (orange blocks) and then re-upsampled again (blue blocks). However, residual connections append the last representation at each scale on the left-hand side to the first representation at the same scale on the right-hand side (orange arrows). The convolutions do not use zero-padding, and so the size decreases slightly with each layer even without downsampling. This means that the representations from the left-hand side must be cropped (dashed squares) before appending to the right-hand side.

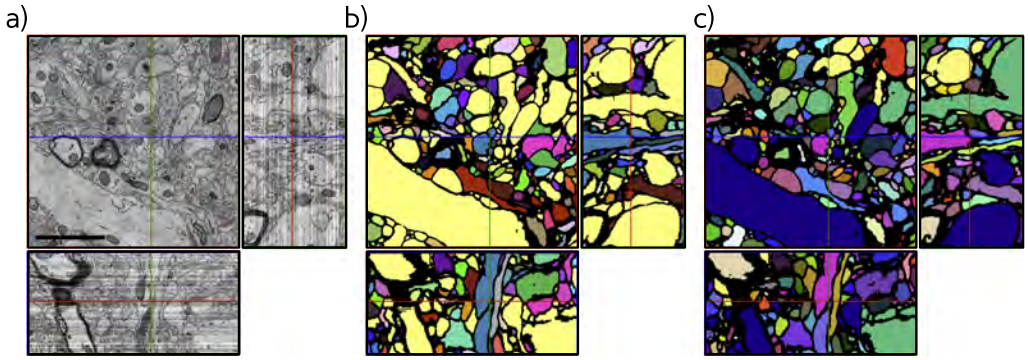

Figure 11.11 Segmentation using U-Net in 3D. a) Three slices through a 3D volume of mouse cortex taken by scanning electron microscope. b) A single U-Net is used to classify voxels as being inside or outside neurites. Connected regions are identified with different colors. c) For a better result, an ensemble of five U-Nets is trained and a voxel is only classified as belonging to the cell if all five networks agree. Adapted from Falk et al. (2019).

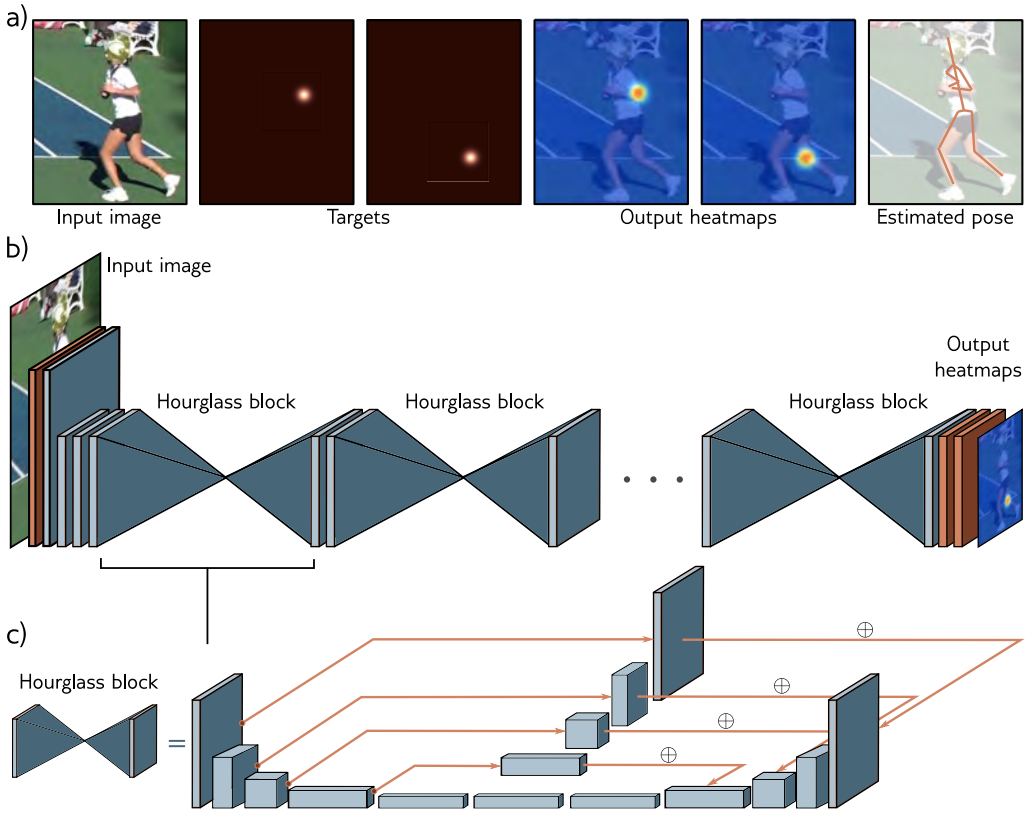

Figure 11.12 Stacked hourglass networks for pose estimation. a) The network input is an image containing a person and the output is a set of heatmaps, where there is one heatmap for each joint. This is formulated as a regression problem where the targets are images with small highlighted regions at the ground-truth joint positions. Each joint position can be extracted by taking the peak of the estimated heatmap and these can be combined to reconstruct the 2D pose. b) The architecture consists of initial convolutional and residual layers followed by a series of hourglass blocks. c) Each hourglass block consists of an encoder-decoder network similar to the U-Net except that the convolutions use zero-padding, some further processing is done in the residual links, and these links add this processed representation rather than concatenate it. Each blue cuboid is itself a bottleneck residual block (figure 11.7b). Adapted from Newell et al. (2016).

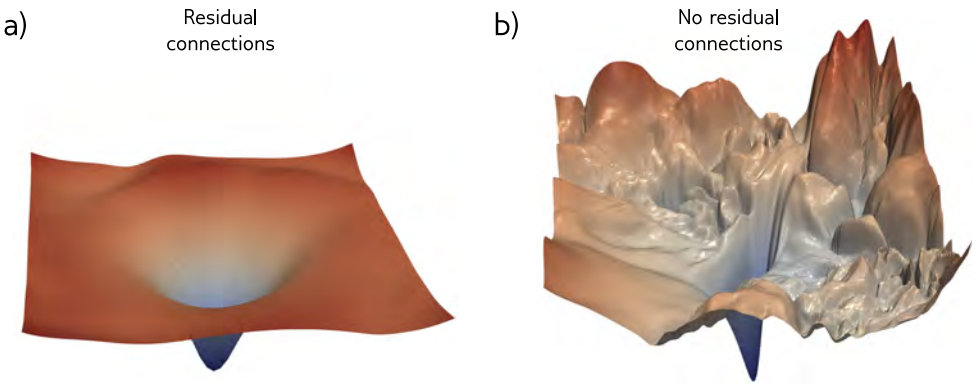

Figure 11.13 Visualizing neural network loss surfaces. Each plot shows the loss surface in two random directions in parameter space around the minimum found by SGD for an image classification task on the CIFAR-10 dataset. These directions are normalized to facilitate side-by-side comparison. a) Residual net with 56 layers. b) Results from the same network without skip connections. The surface is clearly smoother with the skip connections. This facilitates learning and makes the final network performance more robust to small errors in the parameters so that it will likely generalize better. Adapted from Li et al. (2018a).

network but two pieces of evidence have emerged that contradict this viewpoint.

First, shallower, wider residual networks sometimes outperform deeper, narrower ones with a comparable parameter count. In other words, better performance can sometimes be achieved with a network with fewer layers but more channels per layer. Second, there is evidence that the gradients during training do not propagate effectively through very long paths in the unraveled network (figure 11.4b). In effect, a very deep network may be acting more like a combination of shallower networks.

The current view is that residual connections add some value of their own, as well as allowing training of deeper networks. This perspective is supported by the fact that the loss surfaces of residual networks around a minimum tend to be smoother and more predictable than those for the same network when the skip connections are removed (figure 11.13). This may make it easier to learn a good solution that generalizes well.

11.7 Summary

Increasing network depth indefinitely causes both training and test performance to decrease for image classification tasks. It is conjectured that this is because the gradient of the loss with respect to the parameters early in the network changes quickly and unpredictably relative to the update step size. Residual connections add the processed representation back to their own input. This means that each layer contributes directly

to the output as well as indirectly, so propagating gradients through many layers is not mandatory and the surface becomes smoother.

Residual networks do not suffer from vanishing gradients but introduce an exponential increase in the variance of the activations during forward propagation through the network, and corresponding problems with exploding gradients. This is usually handled by adding batch normalization layers to the network. These normalize by the empirical mean and variance of the batch and then shift and rescale using parameters that are learned. If these parameters are initialized judiciously, then very deep networks can be trained. There is evidence that both residual links and batch normalization make the loss surface smoother, which permits larger learning rates. Moreover, the variability in the batch statistics adds a source of regularization.

Residual blocks have been incorporated into convolutional networks. They allow deeper networks to be trained with commensurate increases in image classification performance. Variations of residual networks include the DenseNet architecture, which concatenates outputs of all prior layers to feed into the current layer, and U-Nets, which incorporate residual connections into encoder-decoder models.

Notes

Residual connections: Residual connections were introduced by He et al. (2016a) who built a network with 152 layers, which was eight times larger than VGG (figure 10.17), and which achieved state-of-the-art performance on the ImageNet classification task. Each residual block consisted of a convolutional layer followed by batch normalization, a ReLU activation, a second convolutional layer, and second batch normalization. A second ReLU function was then applied after this block was added back to the main representation. This architecture was termed *ResNet v1*. He et al. (2016b) investigated different variations of residual architectures, in which either (i) processing could also be applied along the skip connection or (ii) after the two branches had recombined. They concluded that neither of these was necessary leading to the architecture in figure 11.7, which is sometimes termed a *pre-activation residual block* and is the backbone of *ResNet v2*. They trained a network with 200 layers that improved further on the ImageNet classification task (see figure 11.8). Since this time, new methods for regularization, optimization, and data augmentation have been developed and Wightman et al. (2021) exploit these to present a more modern training pipeline for the ResNet architecture.

Why residual connections help: Residual networks certainly allow deeper networks to be trained. Presumably, this is related to reducing shattered gradients (Balduzzi et al. 2017) at the start of training and the smoother loss surface near the minima as depicted in figure 11.13 (Li et al. 2018a). Residual connections alone (i.e., without batch normalization) increase the trainable depth of a network by roughly a factor of two (Sankararaman et al. 2020). With batch normalization, very deep networks can be trained, but it is unclear that depth is critical for performance. Zagoruyko & Komodakis (2016) showed that wide residual networks with only 16 layers outperformed all residual networks of the time for image classification. Orhan & Pitkow (2017) propose a different explanation for why residual connections improve learning in terms of eliminating singularities (places on the loss surface where the Hessian is degenerate).

Related architectures: Residual connections are a special case of *highway networks* (Srivastava et al. 2015) which also split the computation into two branches and additively recombine. Highway networks use a gating function that weights the inputs to the two branches in a way

that depends on the data itself, whereas residual networks send the data down both branches in a straightforward manner. Xie et al. (2017) introduced the ResNeXt architecture, which places a residual connection around multiple parallel convolutional branches.

Neural ODEs: Each layer of a residual network makes an additive change to the representation. One can consider this as a series of time steps, in which the representation is sequentially modified. *Neural ordinary differential equations* or *neural ODEs* (Chen et al. 2018b) make these timesteps infinitesimal. Instead of multiple layers, there is now a single network that takes the current representation and the time as inputs and returns the change (derivative) of the representation at that time. This allows tools designed for ODEs to be exploited in the forward and backward passes. Grathwohl et al. (2018) proposed FFJORD which extended this idea to develop reversible networks based on ODEs (see chapter 16). Dupont et al. (2019) noted that networks that make continuous changes cannot represent all functions and proposed ANODE which is a simple technique that augments the representation space to remedy this problem. Neural ODEs are a special case of a more general construction called an *implicit layer* (see Kolter et al. 2020).

Residual networks as ensembles: Veit et al. (2016) first characterized residual networks as ensembles of shorter networks and depicted the "unraveled network" interpretation (figure 11.4b). They provide evidence that this interpretation is valid by showing that deleting layers in a trained network (and hence a subset of paths) only has a modest effect on performance. Conversely, removing a layer has catastrophic consequences for a purely sequential network like VGG. They also looked at the gradient magnitudes along paths of different lengths and showed that the gradient vanishes in longer paths. In a residual network consisting of 54 blocks, almost all of the gradient updates during training were from paths of length 5 to 17 blocks long, even though these only constitute 0.45% of the total paths. In other words, the effect of adding more blocks seems to be mainly to add more parallel shorter paths rather than to create a network that is truly deeper.

Regularization for residual networks: L2 regularization has a fundamentally different effect in vanilla networks and residual networks without BatchNorm. In the former, it encourages the output of the layer to be a constant function determined by the bias. In the latter, it encourages the output of the residual block to compute the identity.

Several regularization methods have been developed that are targeted specifically at residual architectures. ResDrop (Yamada et al. 2016), stochastic depth (Huang et al. 2016), and RandomDrop (Yamada et al. 2019) all regularize residual networks by randomly dropping residual blocks during the training process. In the latter case, the propensity for dropping a block is determined by a Bernouilli variable, whose parameter is linearly decreased during training. At test time, the residual blocks are added back in with their expected probability. These methods are effectively versions of Dropout, in which all the hidden units in a block are simultaneously dropped in concert. In the multiple paths view of residual networks (figure 11.4b), they simply remove some of the paths at each training step. Wu et al. (2018) developed BlockDrop which analyzes an existing network and decides which residual blocks to use at runtime with the goal of improving the efficiency of inference.

Separate regularization procedures have been developed for networks like ResNeXt which have multiple paths inside the residual block. Shake-shake (Gastaldi 2017a; Gastaldi 2017b) randomly re-weights these paths differently during the forward and backward passes. In the forward pass, this can be viewed as synthesizing random data, and in the backward pass as injecting another form of noise into the training method. ShakeDrop (Yamada et al. 2019) also draws a Bernoulli variable that decides whether each block will be subject to Shake-Shake or behave like a normal residual unit on this training iteration.

Batch normalization: Batch normalization was introduced by Ioffe & Szegedy (2015) outside of the context of residual networks. They showed empirically that it allowed higher learning rates, increased speed of convergence, and made sigmoid activation functions more practical (since the distribution of outputs is controlled, and so examples are less likely to fall in the saturated extremes of the sigmoid). Balduzzi et al. (2017) investigated the activation of hidden units in later layers of deep networks with ReLU functions at initialization. They showed that many such hidden units were always active or always inactive regardless of the input, but that BatchNorm reduced this tendency.

Although batch normalization helps stabilize the forward propagation of signals through a network, Yang et al. (2019) showed that it causes gradient explosion in ReLU networks without skip connections, with each layer increasing the magnitude of the gradients by $\sqrt{\pi/(\pi-1)} \approx 1.21$. This argument is summarized by Luther (2020). Since a residual network can be seen as a combination of paths of different lengths (figure 11.4), this effect must be present in residual networks as well. Presumably, however, the benefit of removing the 2^K increases in magnitude in the forward pass of a network with K layers outweighs the harm done by increasing the gradients by 1.21^K in the backward pass, and so overall BatchNorm makes training more stable.

Problem 11.12

Variations of batch normalization: Several variants of BatchNorm have been proposed (figure 11.14). BatchNorm normalizes each channel separately based on statistics gathered across the batch. *Ghost batch normalization* or *GhostNorm* (figure 11.14b) uses only part of the batch to compute the normalization statistics, which makes them noisier and increases the amount of regularization when the batch size is very large.

When the batch size is very small, or the fluctuations within a batch are very large (as is often the case in natural language processing tasks), the statistics in BatchNorm may become unreliable. Ioffe (2017) proposed *batch renormalization*, which keeps a running average of the batch statistics and modifies the normalization of any batch to ensure that it is more representative. A second problem with batch normalization is that it is unsuitable for use in recurrent neural networks (networks for processing sequence,s in which the last output is fed back as an additional input as we move through the sequence — see figure 12.18). This is because the statistics must be stored at each step in the sequence, and it's not clear what to do if a test sequence is longer than the training sequences. A third problem is that batch normalization needs access to the whole batch, but this may not be easily available if the training is distributed across several machines using data parallelism.

Layer normalization or *LayerNorm* (Ba et al. 2016) avoids using the batch statistics by normalizing each data example separately, using statistics gathered across the channels and spatial position (figure 11.14c). However, there is still a separate learned scale γ and offset δ per channel. *Group normalization* or *GroupNorm* (Wu & He 2018) is similar to LayerNorm but divides the channels into groups and computes the statistics for each group separately, computing statistics across the within-group channels and spatial position (figure 11.14d). Again, there are still separate scale and offset parameters per channel. *Instance normalization* or *InstanceNorm* (Ulyanov et al. 2016) takes this to the extreme where the number of groups is the same as the number of channels, and so each channel is normalized separately (figure 11.14e), using statistics gathered across spatial position alone. Salimans & Kingma (2016) investigated normalizing the network weights rather than the activations, but this has been less empirically successful. Teye et al. (2018) introduced *Monte Carlo batch normalization*, which can provide meaningful estimates of uncertainty in the predictions of neural networks. A recent comparison of the properties of different normalization schemes can be found in Lubana et al. (2021).

Why BatchNorm helps: BatchNorm helps control the initial gradients in a residual network (figure 11.6c). However, the mechanism by which BatchNorm improves performance more generally is not well understood. The stated goal of Ioffe & Szegedy (2015) was to reduce problems caused by *internal covariate shift*, which is the change in the distribution of inputs to a layer caused by updating preceding layers during the backpropagation update. However,

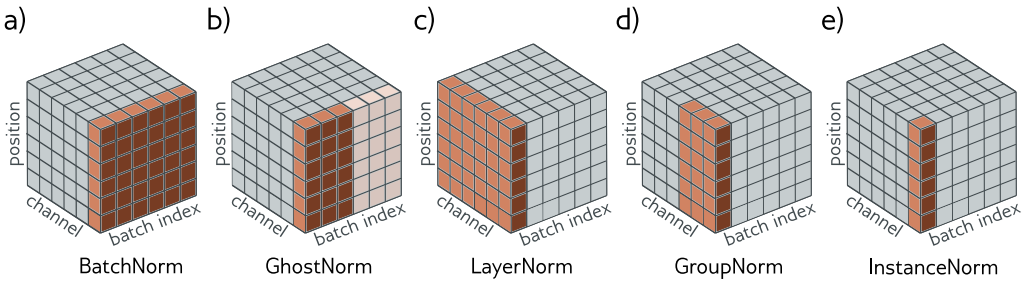

Figure 11.14 Normalization schemes. BatchNorm modifies each channel separately, but modifies each member of the batch in the same way based on statistics gathered across the batch and spatial position. Ghost BatchNorm computes these statistics from only part of the batch to make them more variable. LayerNorm computes statistics for each member of the batch separately, based on statistics gathered across the channels and spatial position. It retains a separate learned scaling factor for each channel. GroupNorm normalizes within each group of channels, and also retains a separate scale and offset parameter for each channel. InstanceNorm normalizes within each channel separately computing the statistics across spatial position only. Adapted from Wu & He (2018).

Santurkar et al. (2018) provided evidence against this view by artificially inducing covariate shift and showing that networks with and without BatchNorm performed equally well.

Motivated by this, they searched for another explanation as to why BatchNorm should improve performance. They showed empirically for the VGG network that adding batch normalization decreases the variation in both the loss and its gradient as we move in the gradient direction. In other words, the loss surface is both smoother and changes more slowly, which is why larger learning rates are possible. They also provide theoretical proofs for both these phenomena and show that for any parameter initialization, the distance to the nearest optimum is less for networks with batch normalization. Bjorck et al. (2018) also argue that BatchNorm improves the properties of the loss landscape and allows larger learning rates.

Other explanations of why BatchNorm improves performance include that it decreases the importance of tuning the learning rate (Ioffe & Szegedy 2015; Arora et al. 2018). Indeed Li & Arora (2019) show that it's possible to use an exponentially increasing learning rate schedule with batch normalization. Ultimately, this is because batch normalization makes the network invariant to the scales of the weight matrices. An intuitive visualization of this effect is provided by Huszár (2019).

Hoffer et al. (2017) identified that BatchNorm has a regularizing effect due to fluctuations in statistics due to the random composition of the batch. They proposed manipulating this directly by using a *ghost batch size*, in which the mean and standard deviation statistics are computed over a subset of the batch. In this way, large batches can be used, without losing the regularizing effect of the additional noise when the batch size is smaller. Luo et al. (2018) investigate the regularization effects of batch normalization.

Alternatives to batch normalization: Although BatchNorm is widely used, it is not strictly necessary to train deep residual nets; there are other ways of making the loss surface tractable. Balduzzi et al. (2017) proposed the rescaling by $\sqrt{1/2}$ in figure 11.6b; they argued that it prevents gradient explosion, but does not resolve the problem of shattered gradients.

Other work has investigated rescaling the output of the function in the residual block before addition back to the input. For example, De & Smith (2020) introduce SkipInit, in which a learnable scalar multiplier is placed at the end of each residual branch. This helps as long as this multiplier is initialized to less than $\sqrt{1/K}$ where K is the number of residual blocks. In practice, they suggest initializing this to zero. Similarly, Hayou et al. (2021) introduce Stable ResNet, which rescales the output of the function in the k^{th} residual block (before addition to the main branch) by a constant λ_k. They prove that in the limit of infinite width, the expected gradient norm of the weights in the first layer is lower bounded by the sum of squares of the scalings λ_k. They investigate setting these to a constant $\sqrt{1/K}$ where K is the number of residual blocks and show that it is possible to train networks with up to 1000 blocks.

Zhang et al. (2019a) introduce a method called *FixUp*, in which every layer is initialized using He normalization, but the last linear/convolutional layer of every residual block is set to zero. This means that the initial forward pass is stable (since each residual block contributes nothing) and the gradients do not explode in the backward pass (for the same reason). They also rescale the branches so that the magnitude of the total expected change in the parameters is constant regardless of the number of residual blocks. Although these methods allow training of deep residual networks, they do not generally achieve the same test performance as when using BatchNorm. This is probably because they do not benefit from the regularization induced by the noisy batch statistics. De & Smith (2020) modify their method to induce regularization via dropout which helps close this gap.

DenseNet and U-Net: DenseNet was first introduced by Huang et al. (2017b), U-Net was developed by Ronneberger et al. (2015), and stacked hourglass networks by Newell et al. (2016). Of these three architectures, U-Net has been most extensively adapted. Çiçek et al. (2016) introduced 3D U-Net and Milletari et al. (2016) introduced VNet, both of which extend U-Net to process 3D data. Zhou et al. (2018) combine the ideas of DenseNet and U-Net in an architecture that both downsamples and re-upsamples the image, but also repeatedly uses intermediate representations. U-Nets are commonly used in medical image segmentation and a review of this work can be found in Siddique et al. (2021). However, they have been applied to other areas including depth estimation (Alhashim & Wonka 2018), semantic segmentation (Iglovikov & Shvets 2018), inpainting (Zeng et al. 2019b), pansharpening (Yao et al. 2018), and image-to-image translation (Isola et al. 2017). U-Nets are also a key component in diffusion models (chapter 17).

Problems

Problem 11.1 Derive equation 11.5 from equations 11.4 by substituting in the expression for $\mathbf{h}_1, \mathbf{h}_2$ and $\mathbf{h}_3$.

Problem 11.2 The network in figure 11.4a has four residual blocks. When we unravel this network, we get one path of length zero, four paths of length one, six paths of length two, four paths of length three, and one path of length four. How many paths of each length would there be if there were (i) three residual blocks and (ii) five residual blocks? Deduce the rule for K residual blocks.

Problem 11.3 Show that the derivative of the network in equation 11.5 with respect to the first layer $\mathbf{f}_1[\mathbf{x}]$ is given by equation 11.6.

Problem 11.4 Consider a residual block, where the block contents comprise a standard linear transformation plus ReLU layer and we have used He initialization. Explain why the distributions of the activations in the block and the skip connection are independent.

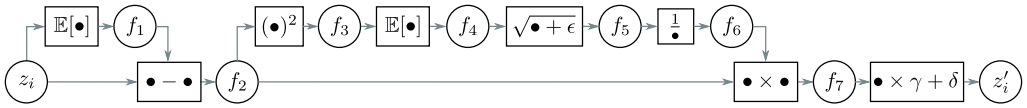

Figure 11.15 Computational graph for batch normalization (see problem 11.10).

Problem 11.5 Write native Python code that shows how the variance of the activations in the forward pass and gradients in the backward pass increases in a residual network as a function of network depth (i.e., similar to figure 7.4). The network should take $D = 100$-dimensional inputs $\mathbf{x}$, which are drawn from a standard normal distribution. The first layer of the network is a linear transform consisting of a 100×100 weight matrix and a 100×1 bias vector. This is followed by 50 residual blocks of the type shown in figure 11.5b and finally another linear transform that maps to a single output f. All parameters are initialized with He initialization. The targets y are also drawn from a standard normal distribution and a least squares cost function is used.

Problem 11.6 Show that adding a normalization factor of $1\sqrt{2}$ after each residual block in your code for problem 11.5 stabilizes the variance of the intermediate values in the forward pass.

Problem 11.7 The forward pass for batch normalization given a batch of scalar values $\{z_i\}_{i=1}^I$ consists of the following operations (figure 11.15):

$$\begin{aligned} f_1 &= \mathbb{E}[z_i] & f_5 &= \sqrt{f_4+\epsilon} \\ f_{2i} &= x_i - f_1 & f_6 &= 1/f_5 \\ f_{3i} &= f_{2i}^2 & f_{7i} &= f_{2i} \times f_6 \\ f_4 &= \mathbb{E}[f_{3i}] & z_i' &= f_{7i} \times \gamma + \delta, \end{aligned} \tag{11.10}$$

where $\mathbb{E}[z_i] = \frac{1}{I}\sum_i z_i$. Write Python code to implement the forward pass.

Now derive the algorithm for the backward pass. Work backward through the computational graph computing the derivatives to generate a set of operations that computes $\partial z_i'/\partial z_i$ for every element in the batch. Write Python code to implement the backward pass.

Problem 11.8 Consider a fully connected neural network with one input, one output, and ten hidden layers, each of which contains twenty hidden units. How many parameters does this network have? How many parameters will it have if we place a batch normalization operation between each linear transformation and ReLU?

Problem 11.9 Consider applying a L2 regularization penalty to the weights in the convolutional layers in figure 11.7a, but not to the scaling parameters of the subsequent BatchNorm layers. What do you expect will happen as training proceeds?

Problem 11.10 Consider a convolutional residual block that contains a batch normalization operation, followed by a ReLU activation function, and then a 3×3 convolutional layer. If the input has 512 channels, then how many parameters are needed to define this block? Now consider a bottleneck residual block that contains three batch normalization / activation / ReLU sequences. The first is a 1×1 convolutional layer that reduces the number of channels from 512 to 128. The second is a 3×3 convolutional layer with the same number of input and output

channels. The third is a 1×1 convolutional layer that increases the number of channels from 128 to 512 (see figure 11.7b). How many parameters are needed to define this block?

Problem 11.11 The DenseNet architecture (figure 11.9) can be described by the equations:

$$\begin{aligned}\mathbf{h}_1 &= \mathbf{f}_1[\mathbf{x}] \\ \mathbf{h}_2 &= \mathbf{f}_2[\mathbf{concat}[\mathbf{x}, \mathbf{h}_1] \\ \mathbf{h}_3 &= \mathbf{f}_3[\mathbf{concat}[\mathbf{x}, \mathbf{h}_1, \mathbf{h}_2] \\ \mathbf{y} &= \mathbf{concat}[\mathbf{x}, \mathbf{h}_1, \mathbf{h}_2, \mathbf{h}_3]\end{aligned} \tag{11.11}$$

Draw this network in the unraveled style (figure 11.4).

Problem 11.12 Figure 7.4 shows that the variance of the activations during forward propagation and the variance of the gradients during backward propagation in vanilla ReLU networks is stabilized by He initialization. Repeat this experiment, but with a BatchNorm layer with $\gamma = 1$ and $\delta = 0$ after each ReLU activation function. Write code to show that the gradients now increase as we move backward through the network at a rate of approximately 1.21 per layer.

Chapter 12

Transformers

Chapter 10 introduced convolutional networks. These are models that are specialized for processing data that lie on a regular grid. They are particularly suited to processing images, which contain a very large number of input variables (precluding the use of fully connected networks) and behave similarly at every position (leading to the idea of parameter sharing).

This chapter introduces transformers; these were originally targeted at natural language processing (NLP) problems, where the network input is a series of high-dimensional embeddings that represent words or word fragments. Language datasets share some of the characteristics of image data. The number of input variables can be very large, and the statistics are similar at every position; it's not sensible to re-learn the meaning of the word dog at every possible position in a body of text. In addition, language datasets have the complication that input sequences are of variable length, and unlike images, there is no way to easily resize them.

12.1 Processing text data

To motivate the transformer, consider the following passage:

The restaurant refused to serve me a ham sandwich because it only cooks vegetarian food. In the end, they just gave me two slices of bread. Their ambiance was just as good as the food and service.

The goal is to design a network that can process this text into a representation that is suitable for downstream tasks. For example, it might be used to classify the review as positive or negative, or answer questions such as "Does the restaurant serve steak?".

We can make three immediate observations. First, the encoded input can be surprisingly large. In this case, each of the 37 words might be represented by an embedding vector of length 1024, and so the input would be of length $37 \times 1024 = 37888$ even for this small passage. A more realistically sized input might have hundreds or even thousands

of words, and so fully connected neural networks are not practical.

Second, one of the defining characteristics NLP problems is that each input (a sentence or multiple sentences) is of a different length; hence, it's not even obvious how to apply a fully connected network. These first two observations both suggest that the network will need to share parameters between the input words in a similar way to the way that a convolutional network shares parameters across different positions in an image.

Third, language is fundamentally ambiguous; it is not clear from the syntax alone that the pronoun it refers to the restaurant and not the ham sandwich. To fully understand the text, the word it should somehow be connected to the word restaurant. In the parlance of transformers, the former word should pay *attention* to the latter. This implies that there must be connections between the words and that the strength of these connections will depend on the words themselves. Moreover, these connections need to extend across large spans of the text; the word their in the last sentence also refers to the restaurant.

12.2 Dot-product self-attention

The previous section argued that a model for processing text will (i) use parameter sharing to cope with long input passages of differing lengths, and (ii) contain connections between word representations that depend on the words themselves. The transformer acquires both properties by using *dot-product self-attention*.

A standard neural network layer $\mathbf{f}[\mathbf{x}]$, takes a $D \times 1$ input $\mathbf{x}$, and applies a linear transformation followed by an activation function $\mathbf{a}[\bullet]$:

$$\mathbf{f}[\mathbf{x}] = \mathbf{a}[\boldsymbol{\beta} + \boldsymbol{\Omega}\mathbf{x}], \tag{12.1}$$

where $\boldsymbol{\beta}$ contains the biases and $\boldsymbol{\Omega}$ contains the weights.

A self-attention block $\mathbf{sa}[\bullet]$ takes N inputs $\mathbf{x}_n$, each of dimension $D \times 1$ and returns N output vectors of the same size. In the context of NLP, each of the inputs $\mathbf{x}_n$ will represent a word or word fragment. First, a set of *values* are computed for each input:

$$\mathbf{v}_n = \boldsymbol{\beta}_v + \boldsymbol{\Omega}_v \mathbf{x}_n, \tag{12.2}$$

where $\boldsymbol{\beta}_v$ and $\boldsymbol{\Omega}_v$ represent biases and weights respectively. Then the n^{th} output $\mathbf{sa}[\mathbf{x}_n]$ is a weighted sum of all the values $\mathbf{v}_n$:

$$\mathbf{sa}[\mathbf{x}_n] = \sum_{m=1}^{N} a[\mathbf{x}_n, \mathbf{x}_m]\mathbf{v}_n. \tag{12.3}$$

The scalar weight $a[\mathbf{x}_n, \mathbf{x}_m]$ is the *attention* that output $\mathbf{x}_n$ pays to input $\mathbf{x}_m$. The N weights $a[\mathbf{x}_n, \mathbf{x}_\bullet]$ are non-negative and sum to one. Hence, self-attention can be thought of as *routing* the values in different proportions to create each output (figure 12.1).

The following sections examine dot-product self-attention in more detail by breaking it down into two parts. First, we'll consider the computation of the values and their subsequent weighting, as described in equation 12.3. Then we'll describe how to compute the attention weights $a[\mathbf{x}_n, \mathbf{x}_m]$.

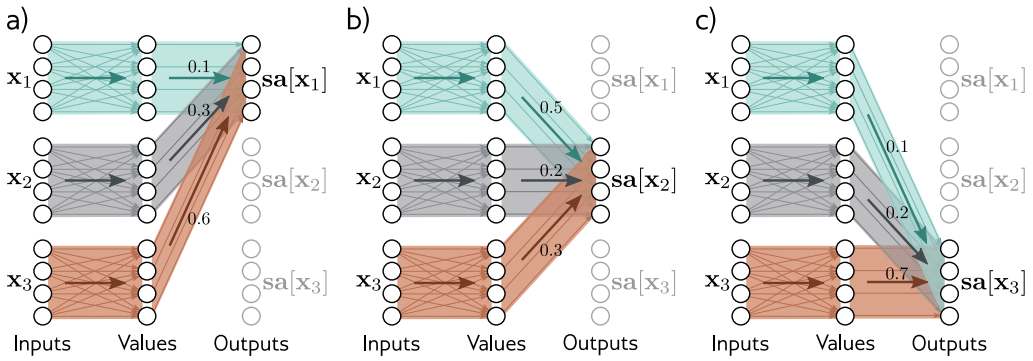

Figure 12.1 Self-attention as routing. The self-attention mechanism takes N inputs $\mathbf{x}_1 \ldots \mathbf{x}_N$, each of size D (here $N=3$ and $D=4$) and processes each separately to compute N sets of values. The n^{th} output $\mathbf{sa}[\mathbf{x}_n]$ is then computed as a weighted sum of the N value vectors, where the weights are positive and sum to one. a) Output $\mathbf{sa}[\mathbf{x}_1]$ is computed as $a[1,1] = 0.1$ times the first values, $a[1,2] = 0.3$ times the second values, and $a[1,3] = 0.6$ times the third values. b) Output $\mathbf{sa}[\mathbf{x}_2]$ is computed in the same way, but this time with weights of 0.5, 0.2, and 0.3. c) The weighting for output $\mathbf{sa}[\mathbf{x}_3]$ is different again. Each output can hence be thought of as a different routing of the N values.

12.2.1 Computing and weighting values

Equation 12.2 shows that the same weights $\boldsymbol{\Omega}_v \in \mathbb{R}^{D\times D}$ and biases $\boldsymbol{\beta}_v \in \mathbb{R}^D$ are applied to each input $\mathbf{x}_m \in \mathbb{R}^D$. This computation scales linearly with the sequence length N, and so requires fewer parameters than a fully connected network relating all DN inputs to all DN outputs. The value computation can be viewed as a sparse matrix operation with shared parameters (figure 12.2b).

The attention weights $a[\mathbf{x}_n, \mathbf{x}_m]$ combine the values from different inputs. They are also sparse in a sense, since there is only one weight for each ordered pair of inputs $(\mathbf{x}_m, \mathbf{x}_n)$, regardless of the size of these inputs (figure 12.2c). It follows that the number of attention weights has a quadratic dependence on the sequence length N, but is independent of the length D of each input $\mathbf{x}_m$.

Problem 12.1

12.2.2 Computing attention weights

In the previous section, we saw that the outputs are the result of two chained linear transformations; the values $\boldsymbol{\beta}_v + \boldsymbol{\Omega}_v\mathbf{x}_m$ are computed independently for each input $\mathbf{x}_m$ and these vectors are combined linearly by the attention weights $a[\mathbf{x}_n, \mathbf{x}_m]$. However, the overall self-attention computation is nonlinear because the attention weights are themselves nonlinear functions of the input. This is an example of a *hypernetwork*, in which one network is used to compute the weights of another.

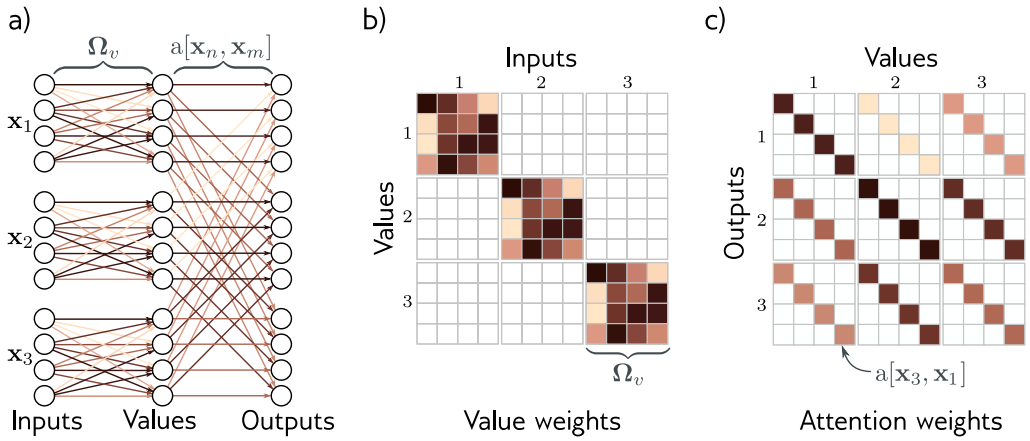

Figure 12.2 Self-attention for $N=3$ inputs $\mathbf{x}_n$, each of which has dimension $D=4$. a) Each input $\mathbf{x}_n$ is operated on independently by the same weights $\boldsymbol{\Omega}_v$ (same color equals same weight) and biases $\boldsymbol{\beta}_v$ (not shown) to form the values $\boldsymbol{\beta}_v+\boldsymbol{\Omega}_v\mathbf{x}_n$. Each output is a linear combination of these values, where there is a single shared attention weight $a[\mathbf{x}_n, \mathbf{x}_m]$ that relates the contribution of the m^{th} value to the n^{th} output. b) Matrix showing block sparsity of linear transformation $\boldsymbol{\Omega}_v$ between inputs and values. c) Matrix showing sparsity of attention weights in the linear transformation relating values and outputs.

To compute the attention, we apply two more linear transformations to the inputs:

$$\begin{aligned}\mathbf{q}_n &= \boldsymbol{\beta}_q + \boldsymbol{\Omega}_q\mathbf{x}_n \\ \mathbf{k}_n &= \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k\mathbf{x}_n,\end{aligned} \tag{12.4}$$

where $\mathbf{q}_n$ and $\mathbf{k}_k$ are referred to as queries and keys, respectively. Then we compute dot products between the queries and keys and pass the results through a softmax function:

$$\begin{aligned}a[\mathbf{x}_n, \mathbf{x}_m] &= \mathrm{softmax}_m\left[\mathbf{k}_m^T\mathbf{q}_n\right] \\ &= \frac{\exp\left[\mathbf{q}_m^T\mathbf{k}_n\right]}{\sum_{m'=1}^{N}\exp\left[\mathbf{k}_{m'}^T\mathbf{q}_n\right]},\end{aligned} \tag{12.5}$$

so for each $\mathbf{x}_m$ they are positive and sum to one (figure 12.3). For obvious reasons, this is known as *dot-product self-attention*.

The names "queries" and "keys" were inherited from the field of information retrieval and have the following interpretation: the dot product operation returns a measure of similarity between its inputs, and so the weights $a[\mathbf{x}_n, \mathbf{x}_\bullet]$ depend on the relative similarities between each query and the keys. The softmax function means that we can

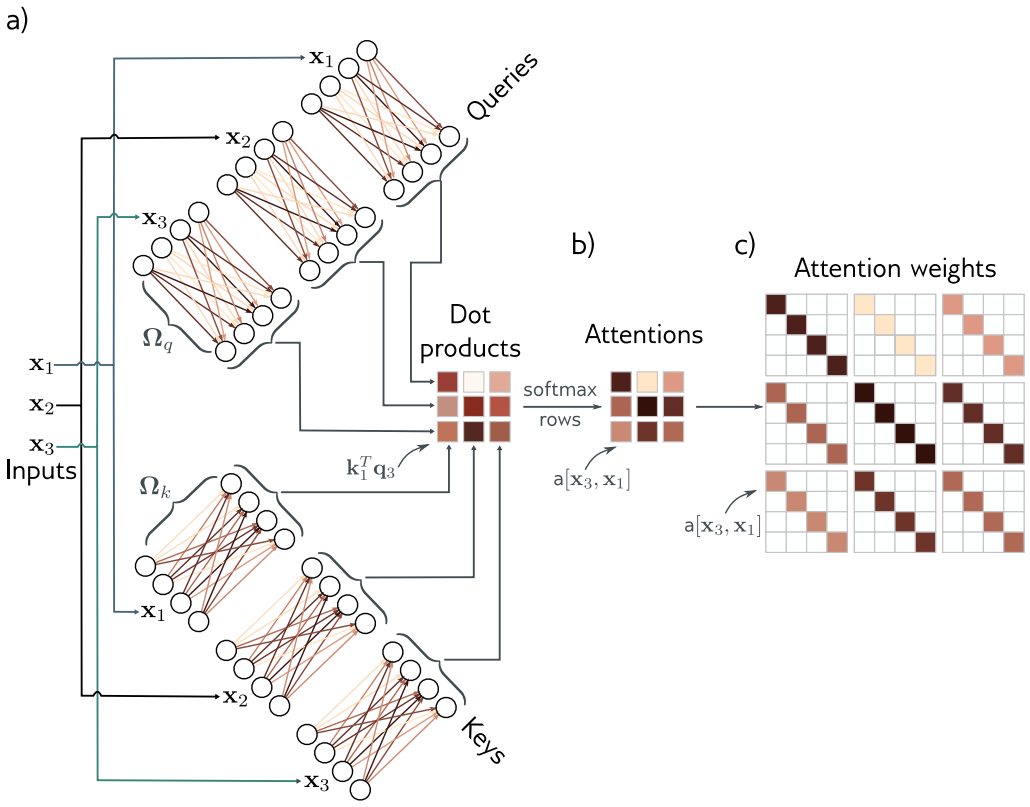

Figure 12.3 Computing attention weights. a) Query vectors $\mathbf{q}_n = \boldsymbol{\beta}_q + \boldsymbol{\Omega}_q\mathbf{x}_n$ and key vectors $\mathbf{k}_n = \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k\mathbf{x}_n$ are computed for each input $\mathbf{x}_m$. b) The dot products between each query and the three keys are passed through a softmax function to form non-negative attentions that sum to one. c) These are used to route the value vectors (figure 12.1) via the sparse matrix from figure 12.2c.

think of the key vectors as "competing" with one another to contribute to the final result. The queries and keys must have the same dimensions. However, these can differ from the dimensions of the values, which are usually the same size as the input so that the representation does not change size.

Problem 12.2

12.2.3 Self-attention summary

The n^{th} output is a weighted sum of the same linear transformation $\boldsymbol{\beta}_v + \boldsymbol{\Omega}_v\mathbf{x}_\bullet$ applied to all of the inputs, where these attention weights are positive and sum to one. The weights depend on a measure of similarity between input $\mathbf{x}_n$ and the other inputs. There is no

activation function, but the mechanism is nonlinear due to the dot-product and softmax operation used to compute the attention weights.

Note that this mechanism fulfills the initial requirements. First, there is a single shared set of of parameters $\boldsymbol{\phi} = \{\boldsymbol{\beta}_v, \boldsymbol{\Omega}_v, \boldsymbol{\beta}_q, \boldsymbol{\Omega}_q, \boldsymbol{\beta}_k, \boldsymbol{\Omega}_k\}$. This is independent of the number of inputs N, and so the network can be applied to different sequence lengths. Second, the connections between the inputs (words) depend on the input representations themselves via the computed attention values.

12.2.4 Matrix form

Problem 12.3

The above computation can be written in a more compact form if we assume that the N inputs $\mathbf{x}_n$ form the columns of the $D \times N$ matrix $\mathbf{X}$. The values, queries, and keys can be computed as:

$$\begin{aligned}\mathbf{V}[\mathbf{X}] &= \boldsymbol{\beta}_v\mathbf{1}^\mathbf{T} + \boldsymbol{\Omega}_\mathbf{v}\mathbf{X} \\ \mathbf{Q}[\mathbf{X}] &= \boldsymbol{\beta}_q\mathbf{1}^\mathbf{T} + \boldsymbol{\Omega}_\mathbf{q}\mathbf{X} \\ \mathbf{K}[\mathbf{X}] &= \boldsymbol{\beta}_k\mathbf{1}^\mathbf{T} + \boldsymbol{\Omega}_\mathbf{k}\mathbf{X},\end{aligned} \tag{12.6}$$

where $\mathbf{1}$ is a $D \times 1$ vector containing ones. The self-attention computation is then:

$$\mathbf{Sa}[\mathbf{X}] = \mathbf{V}[\mathbf{X}] \cdot \mathbf{Softmax}\Big[\mathbf{K}[\mathbf{X}]^T\mathbf{Q}[\mathbf{X}]\Big], \tag{12.7}$$

where the function $\mathbf{Softmax}[\bullet]$ takes a matrix and performs the softmax operation independently on each of its columns (figure 12.4). In this formulation, we have explicitly included the dependence of the values, queries, and keys on the input $\mathbf{X}$ to emphasize that the self-attention computes a kind of triple product based on the inputs. However, from now on we will drop this dependence and just write:

$$\mathbf{Sa}[\mathbf{X}] = \mathbf{V} \cdot \mathbf{Softmax}[\mathbf{K}^T\mathbf{Q}]. \tag{12.8}$$

12.3 Extensions to dot-product self-attention

In the previous section, we described the dot-product self-attention mechanism. Here, we introduce three extensions that are almost always used in practice.

12.3.1 Positional encoding

Observant readers will have noticed that the above mechanism loses some important information; the computation is the same regardless of the order of the inputs $\mathbf{x}_n$. However, when the inputs correspond to the words in a sentence, the order is important.

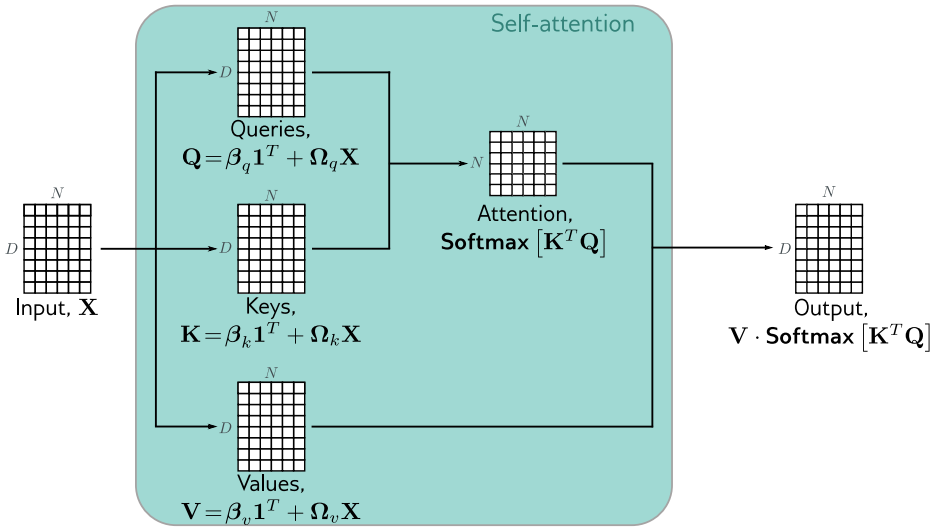

Figure 12.4 Self-attention in matrix form. Self-attention can be implemented efficiently if we store the N input vectors $\mathbf{x}_n$ in the columns of the $D\times N$ matrix $\mathbf{X}$. The input $\mathbf{X}$ is operated on separately by the query matrix $\mathbf{Q}$, key matrix $\mathbf{K}$, and value matrix $\mathbf{V}$. The dot products are then computed using matrix multiplication and a softmax operation is applied independently to each column of the resulting matrix to compute the attentions. Finally, the values are post-multiplied by the attentions to create an output of the same size as the input.

The sentence The woman ate the raccoon has a quite different meaning to The racoon ate the woman. There are two main approaches to incorporating position information.

Absolute position embeddings: A matrix $\boldsymbol{\Pi}$ is added to the input $\mathbf{X}$ that encodes positional information. Each column of $\boldsymbol{\Pi}$ is unique, and so contains information about the position in the input sequence. This matrix may either be chosen by hand or learned. It may be added to the network inputs, or added at every network layer. Sometimes it is added to $\mathbf{X}$ in the computation of the queries and keys, but not for the values.

Relative position embeddings: The input to a self-attention mechanism may be an entire sentence, many sentences, or just a fragment of a sentence, and the absolute position of a word is much less important than the relative position between two inputs. Of course, this can be recovered if the system knows the absolute position of both, but relative position embeddings encode this information directly. Each element of the attention matrix corresponds to a particular offset between query position a and key position b. Relative position embeddings learn a parameter $\pi_{a,b}$ for each offset and use this to modify the attention matrix by adding these values, multiplying by them, or using them to modify the attention matrix in some other way.

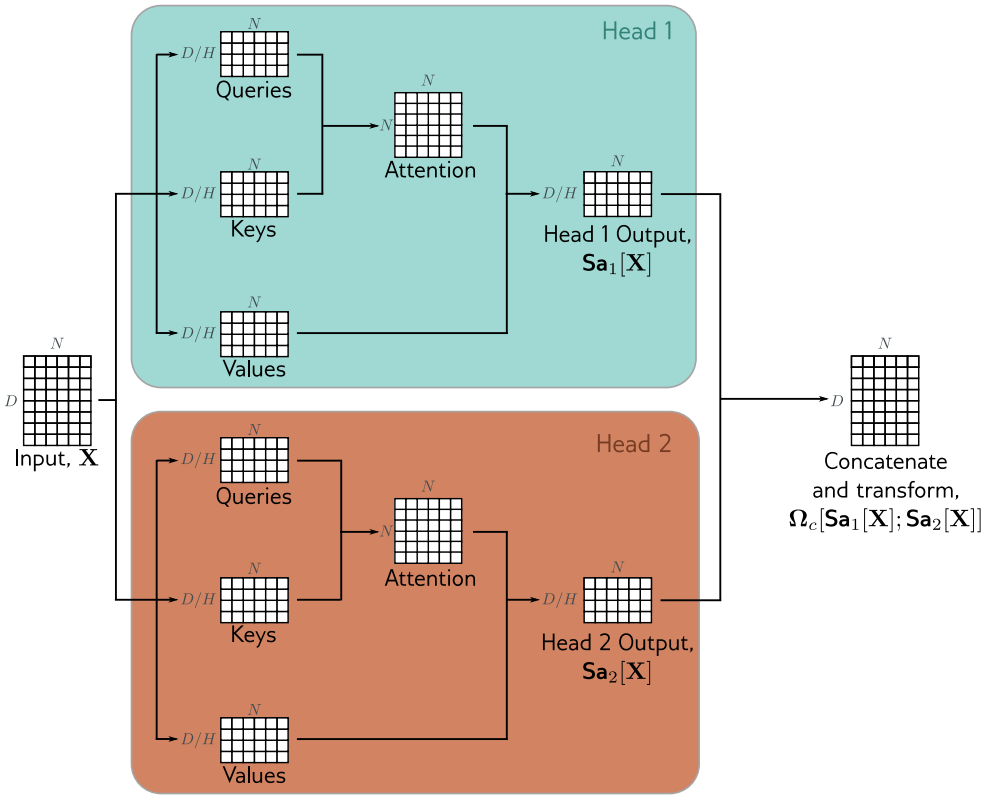

Figure 12.5 Multi-head self-attention. Self-attention occurs in parallel across multiple "heads". Each has its own queries, keys, and values. Here two heads are depicted, in the cyan and orange boxes, respectively. The outputs are vertically concatenated and another linear transformation $\boldsymbol{\Omega}_c$ is applied to recombine them.

12.3.2 Scaled dot product self-attention

Problem 12.4

The dot products in the attention computation can have very large magnitudes and move the arguments to the softmax function into a region where the largest value completely dominates. Now small changes to the inputs to the softmax function have little effect on the output (i.e., the gradients are very small) and the model becomes hard to train. To prevent this, the dot products are scaled by the square root of the dimension D_q of the queries and keys (i.e., the number of rows in $\boldsymbol{\Omega}_q$ and $\boldsymbol{\Omega}_k$, which must be the same):

$$\mathbf{Sa}[\mathbf{X}] = \mathbf{V} \cdot \mathbf{Softmax}\left[\frac{\mathbf{K}^T\mathbf{Q}}{\sqrt{D_q}}\right]. \tag{12.9}$$

This is known as *scaled dot product self-attention*.

12.3.3 Multiple heads

Multiple self-attention mechanisms are usually applied in parallel, and this is known as *multi-head self-attention*. Now H different sets of values, keys, and queries are computed:

$$\begin{aligned}\mathbf{V}_h &= \boldsymbol{\beta}_{vh}\mathbf{1}^\mathbf{T} + \boldsymbol{\Omega}_\mathbf{vh}\mathbf{X} \\ \mathbf{Q}_h &= \boldsymbol{\beta}_{qh}\mathbf{1}^\mathbf{T} + \boldsymbol{\Omega}_\mathbf{qh}\mathbf{X} \\ \mathbf{K}_h &= \boldsymbol{\beta}_{kh}\mathbf{1}^\mathbf{T} + \boldsymbol{\Omega}_\mathbf{kh}\mathbf{X}. \end{aligned} \tag{12.10}$$

The h^{th} self-attention mechanism or *head* can be written as:

$$\mathbf{Sa}_h[\mathbf{X}] = \mathbf{V}_h \cdot \mathbf{Softmax}\left[\frac{\mathbf{K}_h^T\mathbf{Q}_h}{\sqrt{D_q}}\right], \tag{12.11}$$

where we have different parameters $\{\boldsymbol{\beta}_{qh}, \boldsymbol{\Omega}_{qh}\}$, $\{\boldsymbol{\beta}_{kh}, \boldsymbol{\Omega}_{kh}\}$ and $\{\boldsymbol{\beta}_{vh}, \boldsymbol{\Omega}_{vh}\}$ for each head. Typically, if the dimension of the inputs $\mathbf{x}_m$ is D and there are H heads, then the values, queries, and keys will all be of size D/H, as this allows for an efficient implementation. The outputs of these self-attention mechanisms are vertically concatenated and another linear transform $\boldsymbol{\Omega}_c$ is applied to combine them (figure 12.5):

Problem 12.5

$$\mathbf{MhSa}[\mathbf{X}] = \boldsymbol{\Omega}_c\left[\mathbf{Sa}_1[\mathbf{X}]; \mathbf{Sa}_2[\mathbf{X}]; \ldots; \mathbf{Sa}_H[\mathbf{X}]\right]. \tag{12.12}$$

Multiple heads seem to be necessary to make the transformer work well. It has been speculated that they make the self-attention network more robust to bad initializations.

12.4 Transformer layers

Self-attention is just one part of a larger *transformer layer*. This consists of a multi-head self-attention unit (which allows the word representations to interact with each other) followed by a fully connected network $\mathbf{mlp}[\mathbf{x}_\bullet]$ (that operates separately on each word). Both units are residual networks (i.e., their output is added back to the original input). In addition, it is typical to add a LayerNorm operation after both the self-attention and fully connected networks. This is similar to BatchNorm but uses statistics across the tokens within a single input sequence to perform the normalization (section 11.4). The complete layer can be described by the following series of operations:

$$\begin{aligned}\mathbf{X} &\leftarrow \mathbf{X} + \mathbf{MhSa}[\mathbf{X}] \\ \mathbf{X} &\leftarrow \mathbf{LayerNorm}[\mathbf{X}] \\ \mathbf{x}_n &\leftarrow \mathbf{x}_n + \mathbf{mlp}[\mathbf{x}_n] \qquad \forall\, n \in \{1 \ldots N\} \\ \mathbf{X} &\leftarrow \mathbf{LayerNorm}[\mathbf{X}], \end{aligned} \tag{12.13}$$

where the column vectors $\mathbf{x}_n$ are separately taken from the full data matrix $\mathbf{X}$. In a real network, the data would pass through a series of these layers.

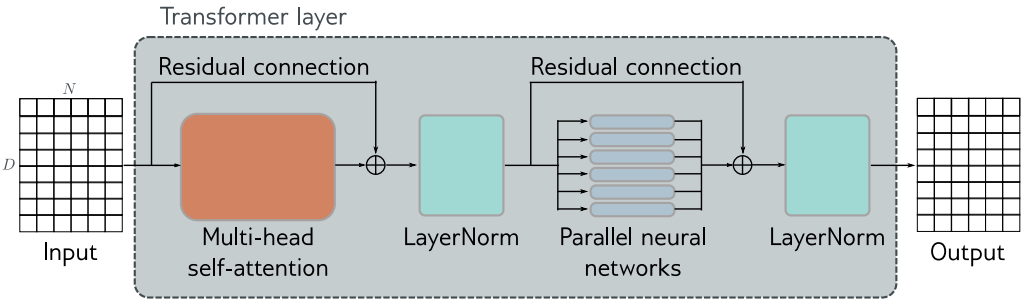

Figure 12.6 The transformer layer. The input consists of a $D \times N$ matrix containing the D dimensional word embeddings for each of the N input tokens. The output is a matrix of the same size. The transformer layer consists of a series of operations. First, there is a multi-head attention block, which allows the word embeddings to interact with one another. This forms the processing of a residual block, so the inputs are added back to the output. Second, a LayerNorm operation is applied. Third, there is a second residual layer where the same two-layer fully connected neural network is applied to each word representation separately. Finally, LayerNorm is applied again.

12.5 Transformers for natural language processing

A typical natural language processing (NLP) pipeline starts with a *tokenize*r that splits the text into words or word fragments. Then each of these tokens is mapped to a learned embedding. These embeddings are passed through a series of transformer layers. We now consider each of these stages in turn.

12.5.1 Tokenization

A text processing pipeline begins with a *tokenizer*. This splits the text into a *vocabulary* of smaller constituent units (tokens) that can be processed by the subsequent network. In the discussion above, we have implied that these tokens represent words, but there are several difficulties with this.

- Inevitably, some words (e.g., names) will not be in the vocabulary.
- It's not clear how to handle punctuation, but this is important. If a sentence ends in a question mark, then we need to encode this information.
- The vocabulary would need different tokens for versions of the same word with different suffixes (e.g., walk, walks, walked, walking) and there is no way to clarify that these variations are related.

One approach would be just to use letters and punctuation marks as the vocabulary, but this would mean splitting text into many very small parts and requiring the subsequent

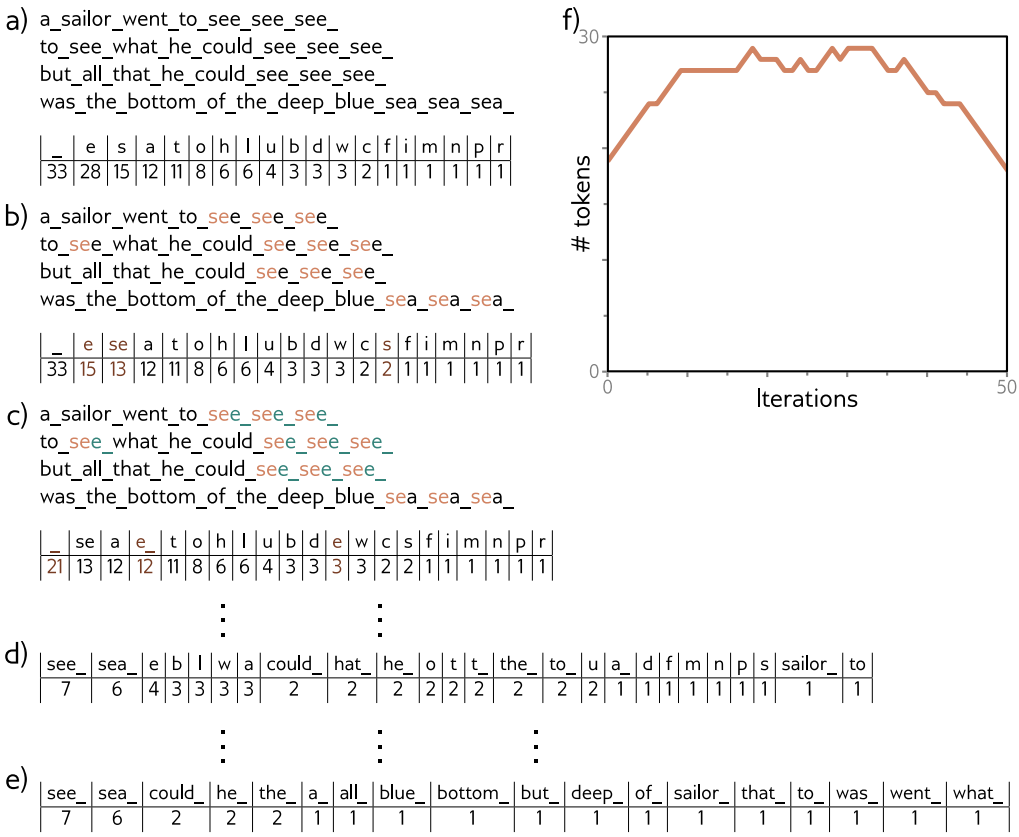

Figure 12.7 Sub-word tokenization. a) A passage of text from a nursery rhyme. The tokens are initially just the characters and whitespace (represented by an underscore), and their frequencies are displayed in the table. b) At each iteration, the sub-word tokenizer looks for the most commonly occurring adjacent pair of characters (in this case se) and merges them. This creates a new token and decreases the counts for the original tokens s and e. c) At the second iteration, the algorithm merges e and the whitespace character_. Note that the last character of the first token to be merged cannot be whitespace, which prevents merging across words. d) After 22 iterations, the tokens consist of a mix of letters, word fragments, and commonly occurring words. e) If we continue this process indefinitely, the tokens just represent the words. f) Over time, the number of tokens increases as we add word fragments to the letters, and then decreases again as we merge these fragments. In a real situation, there would be a very large number of words and the algorithm would terminate when the vocabulary size (number of tokens) reached a predetermined value. Punctuation and capital letters would also be treated as separate input characters.

network to re-learn the relations between them.

In practice, a compromise between using letters and full words is used, and the final vocabulary includes both common words and word fragments from which larger and less frequent words can be composed. The vocabulary is computed using a *sub-word tokenizer* such as *byte pair encoding* (figure 12.7) that greedily merges commonly occurring substrings based on their frequency.

Problem 12.6

12.5.2 Embeddings

Each token in the vocabulary $\mathcal{V}$ is mapped to a *word embedding*. Importantly, the same token always maps to the same embedding. To accomplish this, the N input tokens are encoded in the matrix $\mathbf{T} \in \mathbb{R}^{|\mathcal{V}| \times N}$, where n^{th} column corresponds to the n^{th} token and is a $|\mathcal{V}| \times 1$ one-hot vector (i.e., a vector where every entry is zero except for the entry corresponding to the token, which is set to one). The embeddings for the whole vocabulary are stored in a matrix $\mathbf{\Omega}_e \in \mathbb{R}^{D \times |\mathcal{V}|}$. The input embeddings are then computed as $\mathbf{X} = \mathbf{\Omega}_e \mathbf{T}$, and $\mathbf{\Omega}_e$ is treated like any other network parameter (figure 12.8). A typical embedding size D is 1024 and a typical total vocabulary size $|\mathcal{V}|$ is 30,000, so even before the main network, there are many parameters in $\mathbf{\Omega}_e$ to learn.

12.5.3 Transformer layers

Finally, the input embedding matrix $\mathbf{X}$ is passed through a series of transformer layers, which we'll refer to as a *transformer model*. There are three types of transformer models. An *encoder* transforms the text into a representation that can support a variety of language tasks. A *decoder* generates a new token that continues the input text. *Encoder-decoder models* are used in *sequence-to-sequence models*, which take one text string and convert them to another (e.g., in machine translation). These three variations are described in sections 12.6–12.8, respectively.

12.6 Encoder model example: BERT

BERT is an encoder model that uses a vocabulary of 30,000 tokens. The tokens are converted to 1024-dimensional word embeddings and passed through 24 transformer layers. Each contains a self-attention mechanism with 16 heads, and for each head, the queries, keys, and values are of dimension 64 (i.e., the matrices $\mathbf{\Omega}_{vh}, \mathbf{\Omega}_{qh}, \mathbf{\Omega}_{kh}$ are of size 1024×64). The dimension of the single hidden layer in the neural network layer of the transformer is 4096. The total number of parameters is ~ 340 million. When BERT was introduced, this was considered large, but it is now orders of magnitude smaller than state-of-the-art models.

Encoder models like BERT exploit transfer learning (section 9.3.6). During *pre-training*, the parameters of the transformer architecture are learned using *self-supervision*

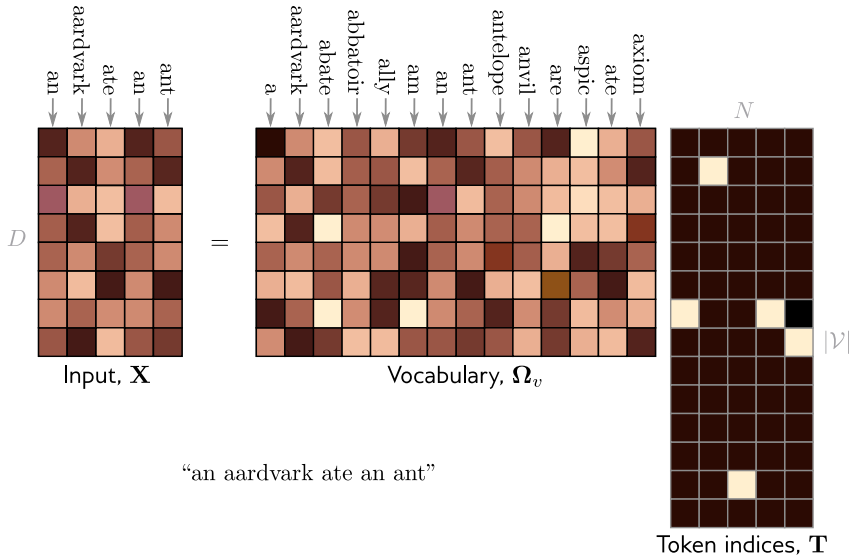

Figure 12.8 The input embedding matrix $\mathbf{X} \in \mathbb{R}^{D \times N}$ contains N embeddings of length D and is created by multiplying a matrix $\boldsymbol{\Omega}_e$ containing the embeddings for the entire vocabulary with a matrix containing one-hot vectors in its columns that correspond to the word or sub-word indices. The vocabulary matrix $\boldsymbol{\Omega}_e$ is considered a parameter of the model and is learned along with the other parameters. Note that the two embeddings for the word an in $\mathbf{X}$ are the same.

from a large corpus of text. The goal here is for the model to learn general information about the statistics of language. In the *fine-tuning stage*, the resulting network is adapted to solve a particular task, using a smaller body of supervised training data.

12.6.1 Pre-training

In the pre-training stage, the network is trained using self-supervision. This allows the use of enormous amounts of data without the need for manual labels. For BERT, the self-supervision task consisted of predicting missing words from sentences from a large internet corpus (figure 12.9).[1] During training, the maximum input length was 512 tokens, and the batch size was 256. The system was trained for a million steps, which corresponded to roughly 50 epochs of the 3.3-billion word corpus.

Problem 12.7

Predicting missing words forces the transformer network to understand some syntax. For example, it might learn that the adjective red is often found before nouns like house or

[1] BERT also used a secondary task that involved predicting whether two sentences were originally adjacent in the text or not but this only marginally improved performance.

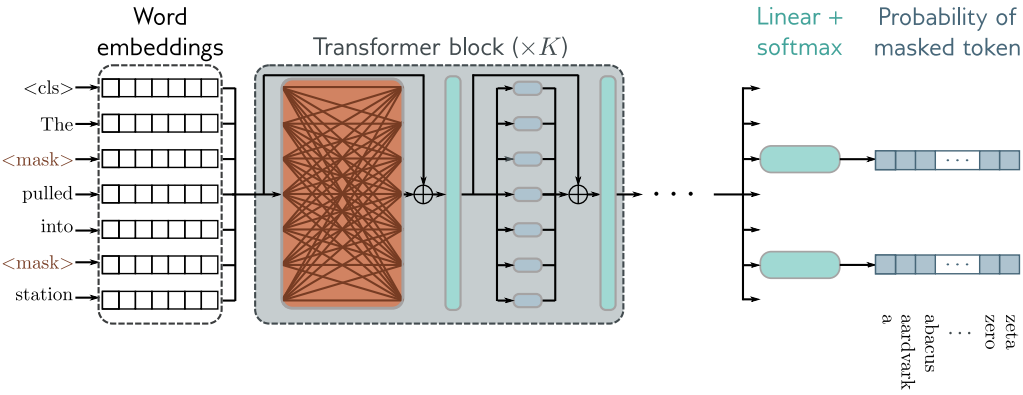

Figure 12.9 Pre-training for BERT-like encoder. The input tokens (and a special <cls> token denoting the start of the sequence) are converted to word embeddings and passed through a series of transformer layers (orange connections indicate that every token attends to every other token in these layers). A small fraction of the input tokens are replaced at random with a generic <mask> token. In pre-training, the goal is to predict the missing word. As such, the output embeddings are passed through a softmax function and the multiclass classification loss (section 5.27) is used. This task has the advantage that it uses both the left and right context to predict the missing word but has the disadvantage that it does not make very efficient use of data; in this case, seven tokens need to be processed to add two terms to the objective function.

car but never before a verb like shout. It also allows the model to learn some superficial *common sense* about the world. For example, after training, the model will assign a higher probability to the missing word train in the sentence The <mask> pulled into the station, than it would to the word peanut. However, the degree of "understanding" that this type of model can ever have is limited.

12.6.2 Fine-tuning

In the fine-tuning stage, the parameters of the model are adjusted to specialize the network to a particular task. An extra layer is appended onto the transformer network to convert the collection of vectors $\mathbf{x}_1, \ldots \mathbf{x}_N$ associated with the input tokens to the desired output format. Examples include:

Text classification: In BERT, there is a special token known as the classification or <cls> token that is placed at the start of each string during pre-training. For text classification tasks like *sentiment analysis* (in which the passage is labeled as having positive or negative emotional tone), the vector associated with the <cls> token is mapped to a single number and passed through a logistic sigmoid (figure 12.10a). This contributes to

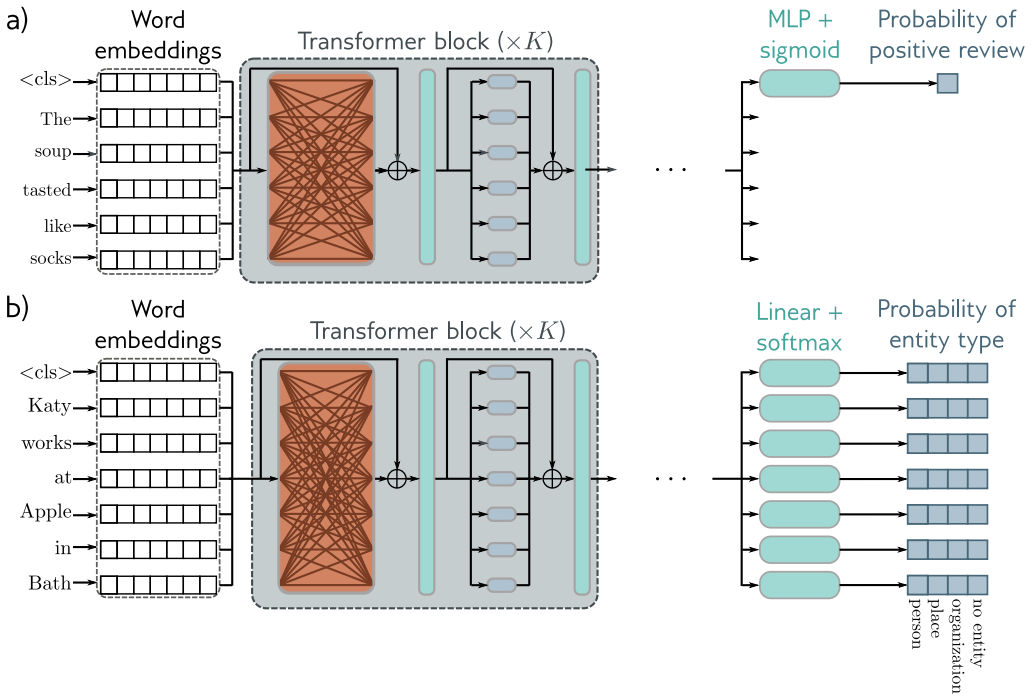

Figure 12.10 After pre-training, the encoder is fine-tuned using manually labeled data to solve a particular task. Usually, a linear transformation or a multi-layer perceptron (MLP) is appended to the encoder to produce whatever output is required for the task. a) Example text classification task. In this sentiment classification task, the <cls> token embedding is used to predict the probability that the review is positive. b) Example word classification task. In this named entity recognition problem, the embedding for each word is used to predict whether the word corresponds to a person, place, or organization, or is not an entity.

a standard binary cross-entropy loss (section 5.5).

Word classification: The goal of *named entity recognition* is to classify each word as an entity type (e.g., person, place, organization, or no-entity). To this end, the vector $\mathbf{x}_n$ associated with each token in the input sequence is mapped to a $K \times 1$ vector where K is the entity type (figure 12.10a). This is passed through a softmax function to create valid probabilities for each class (figure 12.10b), which contribute to a standard multiclass cross-entropy loss.

Text span prediction: In the SQuAD 1.1 question answering task, the question and a passage from Wikipedia containing the answer are concatenated and form the input into the system. BERT is then used to predict the text span in the passage that contains

the answer. Each token associated with the Wikipedia passage maps to two numbers that indicate how likely it is that the text span begins and ends at this location. The resulting two sets of numbers are put through two softmax functions and the probability of any text span being the answer can then be derived by combining the probability of starting and ending at the appropriate places.

12.7 Decoder model example: GPT3

In this section, we present a high-level description of GPT3, which is an example of a decoder model. The basic architecture is extremely similar to the encoder model in that it consists of a series of transformer layers that operate on learned word embeddings. However, the goal is different. The encoder aimed to build a representation of the text that could be fine-tuned to solve a variety of more specific NLP tasks. Conversely, the decoder has one purpose, which is to generate the next token in a sequence. By feeding the extended sequence back into the model, it can generate a coherent text passage.

12.7.1 Language modeling

More formally, GPT3 constructs an autoregressive language model. For any sentence, it aims to model the joint probability $Pr(t_1, t_2, \ldots t_N)$ of the N observed tokens and it does this by factorizing this joint probability into an autoregressive sequence:

$$Pr(t_1, t_2, \ldots t_N) = Pr(t_1) \prod_{n=2}^{N} Pr(t_n | t_1 \ldots t_{n-1}). \tag{12.14}$$

This is easiest to understand with a concrete example. Consider the sentence It takes great personal courage to let yourself appear weak. For simplicity, let's assume that the tokens are the full words. The probability of the full sentence is:

$$\begin{aligned} &Pr(\text{It takes great personal courage to let yourself appear weak}) \quad = \\ &\quad Pr(\text{It}) \times Pr(\text{takes}|\text{It}) \times Pr(\text{great}|\text{It takes}) \times Pr(\text{courage}|\text{It takes great}) \times \\ &\quad Pr(\text{to}|\text{It takes great courage}) \times Pr(\text{let}|\text{It takes great courage to}) \times \\ &\quad Pr(\text{yourself}|\text{It takes great courage to let}) \times \\ &\quad Pr(\text{appear}|\text{It takes great courage to let yourself}) \times \\ &\quad Pr(\text{weak}|\text{It takes great courage to let yourself appear}). \end{aligned} \tag{12.15}$$

This demonstrates the connection between the probabilistic formulation of the loss function and the next token prediction task.

12.7.2 Masked self-attention

To train a decoder, we maximize the log-probability of the input text under the autoregressive model. Ideally, we would pass in the whole sentence and compute all of the log probabilities and gradients simultaneously. However, this poses a problem; if we pass in the full sentence, then the term computing $\log[Pr(\text{great}|\text{It takes})]$ has access to both the answer great and the right context courage to let yourself appear weak.

Fortunately, in a transformer network, the tokens only interact in the self-attention layers. Hence, the problem can be resolved by ensuring that the attention to the answer and the right context is zero. This can be achieved by setting the corresponding dot products in the self-attention computation (equation 12.5) to negative infinity before they are passed through the $\textbf{softmax}[\bullet]$ function. This is known as *masked self-attention*.

The full decoder network operates as follows. The input text is tokenized, and the tokens are converted to embeddings. The embeddings are passed into the transformer network, but now the transformer layers use masked self-attention so that they can only attend to the current and previous tokens. Each of the output embeddings can be thought of as representing a partial sentence, and for each the goal is to predict the next token in the sequence. Consequently, after the transformer layers, a linear layer maps each word embedding to the size of the vocabulary, followed by a $\textbf{softmax}[\bullet]$ function that converts these values to probabilities. We aim to maximize the sum of the log probabilities of the next token in the ground truth sequence at every position using a standard multiclass cross-entropy loss (figure 12.11).

12.7.3 Generating text from a decoder

The autoregressive language model is the first example of a *generative model* discussed in this book. Since it defines a probability model over text sequences, it can be used to sample new examples of plausible text. To generate from the model, we start with an input sequence of text (which might be just a special <start> token) and feed this into the network which then outputs the probabilities over possible next tokens. We can then either pick the most likely token or sample from this probability distribution. The new extended sequence can be fed back into the decoder network that outputs the probability distribution over the next token and in this way, we can generate large bodies of text. The computation can be made quite efficient as prior embeddings do not depend on subsequent ones due to the masked self-attention and hence much of the earlier computation can be recycled as we generate subsequent tokens.

In practice, many strategies that can be employed to help make the output text more coherent. For example, *beam search* keeps track of multiple possible sentence completions with the aim of finding the overall most likely (which is not necessarily found by greedily choosing the most likely next word at each step). *Top-k sampling* randomly draws the next word from only the top-K most likely possibilities to prevent the system from accidentally choosing from the long tail of low probability tokens and leading to an unnecessary linguistic dead end.

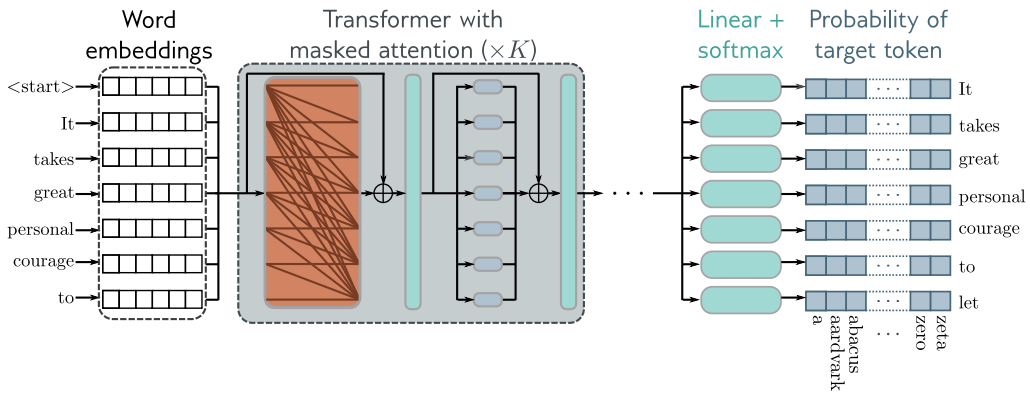

Figure 12.11 GPT3-type decoder network. The tokens are mapped to word embeddings with a special <start> token at the beginning of the sequence. The embeddings are passed through a series of transformers that use masked self-attention. Here, each position in the sentence can only attend to its own embedding and the embeddings of tokens earlier in the sequence (orange connections). The goal at each position is to maximize the probability of the ground truth token that follows in the sequence. In other words, at position one we want to maximize the probability of the token It, at position two we want to maximize the probability of the token takes, and so on. The masked self-attention is designed to ensure that the system cannot cheat by looking at subsequent inputs. This system has the advantage that it makes efficient use of the data since every word contributes a term to the loss function. However, it has the disadvantage that it only uses the left context of each word to predict it.

12.7.4 GPT3 and few-shot learning

GPT3 applies these ideas on a massive scale. The sequence lengths are 2048 tokens long and since multiple spans of 2048 tokens are processed at once, the total batch size is 3.2 million tokens. There are 96 transformer layers (some of which implement a sparse version of attention), each of which processes a word embedding of size 12288. There are 96 heads in the self-attention layers and the value, query, and key dimension is 128. It is trained with 300 billion tokens and learns a total of 175 billion parameters.

Here's an example of completing text from the GPT3 model where the text provided to the model is in orange and the generated text is in blue:

Understanding Deep Learning is a new textbook from MIT Press by Simon Prince that's designed to offer an accessible, broad introduction to the field. Deep learning is a branch of machine learning that is concerned with algorithms that learn from data that is unstructured or unlabeled. The book is divided into four sections:

1. Introduction to deep learning
2. Deep learning architecture

3. Deep learning algorithms
4. Applications of deep learning

The first section offers an introduction to deep learning, including its history and origins. The second section covers deep learning architecture, discussing various types of neural networks and their applications. The third section dives into deep learning algorithms, including supervised and unsupervised learning, reinforcement learning, and more. The fourth section applies deep learning to various domains, such as computer vision, natural language processing, and robotics.

This continuation is plausible, if not entirely accurate.

One surprising property of learning a model on this scale is that it can perform many tasks without the need for fine-tuning. If we provide several examples of correct question/answer pairs, and then another question, it can often answer the final question correctly just by completing the sequence. One example of this is correcting English grammar:

Poor English input: I eated the purple berries.
Good English output: I ate the purple berries.
Poor English input: Thank you for picking me as your designer. I'd appreciate it.
Good English output: Thank you for choosing me as your designer. I appreciate it.
Poor English input: The mentioned changes have done. or I did the alteration that you requested. or I changed things you wanted and did the modifications.
Good English output: The requested changes have been made. or I made the alteration that you requested. or I changed things you wanted and made the modifications.
Poor English input: I'd be more than happy to work with you in another project.
Good English output: I'd be more than happy to work with you on another project.

Here, the text containing the paired examples in brown was provided as context for GPT3 and the system then generated the correct answer in blue. This phenomenon extends to many situations including generating code snippets based on natural language descriptions, arithmetic, translating between languages, and answering questions about text passages. Consequently, it is argued that enormous language models are *few-shot learners*; they can learn to do novel tasks based on just a few examples. However, in practice performance is erratic, and it is not clear the extent to which it is extrapolating from learned examples rather than merely interpolating, or copying verbatim.

12.8 Encoder-decoder model example: machine translation

Translation between languages is an example of a *sequence-to-sequence* task. This requires an encoder (to compute a good representation of the source sentence) and a decoder (to generate the sentence in the target language). This task can be tackled using an *encoder-decoder* model.

Consider the example of translating from English to French. The encoder receives the sentence in English and processes it through a series of transformer layers to create an

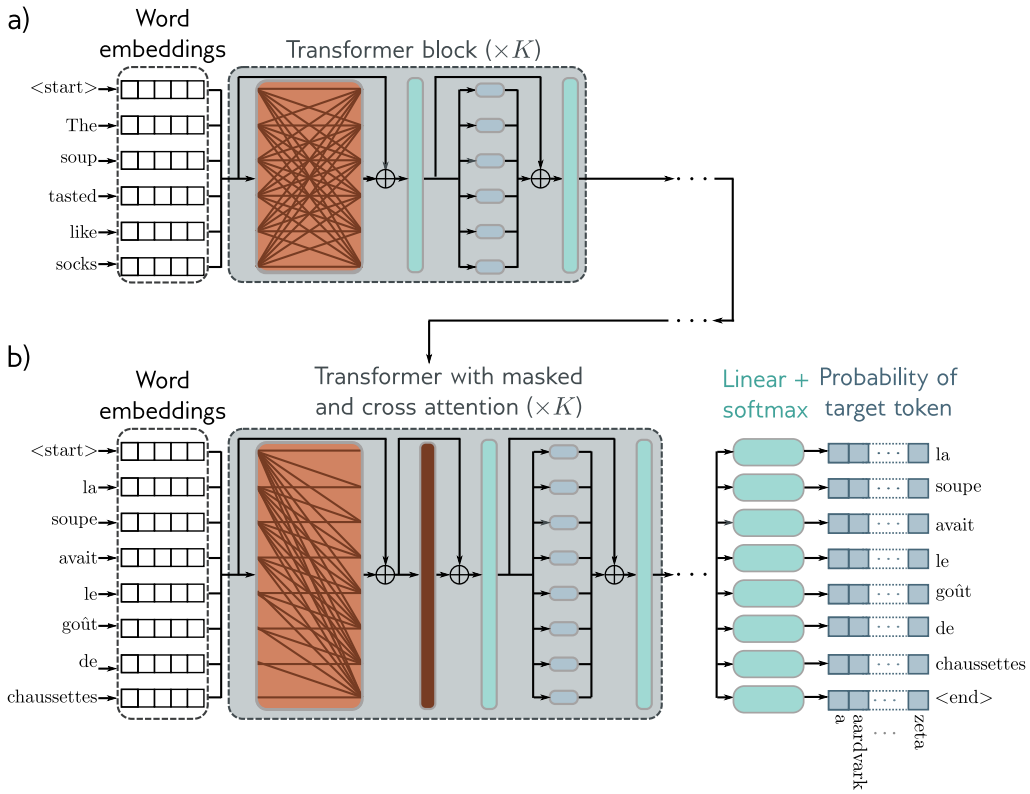

Figure 12.12 Encoder-decoder architecture. Two sentences are input to the system and with the goal of learning to translate the first into the second. The first sentence is passed through a standard encoder. The second sentence is passed through a decoder that uses masked self-attention but also attends to the output embeddings of the encoder using cross attention (brown rectangle). The loss function is the same as for the decoder; we want to maximize the probability of the next word in the output sequence.

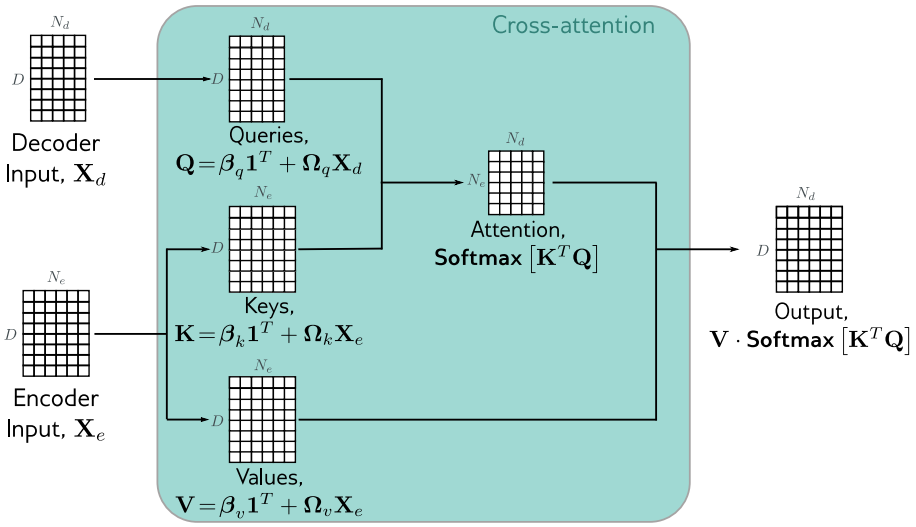

Figure 12.13 Cross attention. The flow of computation is the same as for standard self-attention. However, the queries are now calculated from the decoder embeddings $\mathbf{X}_d$, and the keys and values from the encoder embeddings $\mathbf{X}_e$.

output representation for each token. During training, the decoder receives the sentence in French and passes it through a series of transformer layers that use masked self-attention and produce the subsequent word at each position. However, the decoder layers also attend to the output of the encoder. Consequently, each French output word is conditioned not only on the previous output words but also on the entire English sentence that it is translating (figure 12.12).

This is achieved by modifying the transformer layers in the decoder. The original transformer layer in the decoder (figure 12.6) consisted of a masked self-attention layer followed by a neural network applied individually to each embedding. A new self-attention layer is added between these two components, in which the decoder embeddings attend to the encoder embeddings. This uses a version of self-attention known as *encoder-decoder attention* or *cross attention* where the queries are computed from the decoder embeddings, and the keys and values from the encoder embeddings (figure 12.13).

12.9 Transformers for long sequences

Since each token in a transformer encoder model interacts with every other token, the computational complexity scales quadratically with the length of the sequence. For a decoder model, each token only interacts with previous tokens, and so there are roughly

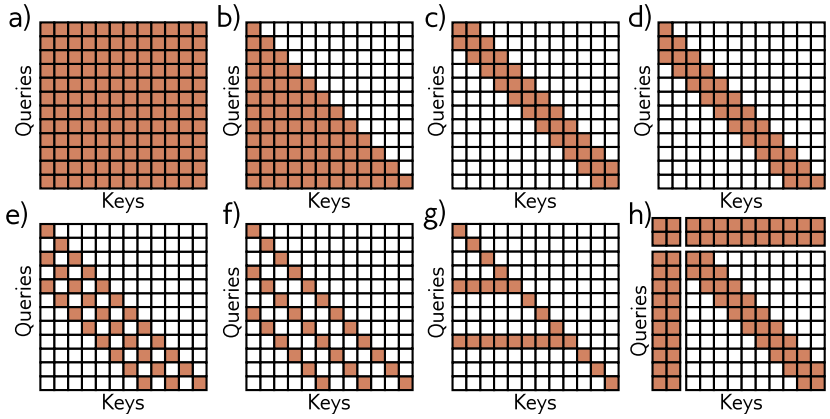

Figure 12.14 Interaction matrices for self-attention. a) In an encoder, every token interacts with every other token and computation expands quadratically with the number of tokens. b) In a decoder, each token only interacts with the previous tokens, but complexity is still quadratic. c) Complexity can be reduced by using a convolutional structure (encoder case) d) Convolutional structure for decoder case. e–f) Convolutional structure with dilation rate of two and three (decoder case). g) Another strategy is to allow selected tokens to interact with all the other tokens (encoder case) or all the previous tokens (decoder case, pictured). h) Finally, new global tokens can be introduced (left two columns and top two rows). These interact with all of the tokens as well as with each other.

half the number of interactions, but the complexity still scales quadratically. These relationships can be visualized as interaction matrices (figure 12.14a–b).

This quadratic increase in the amount of computation ultimately limits the length of sequences that can be used. Many methods have been developed to extend the transformer to cope with longer sequences. An important subset of these prunes the self-attention interactions, or equivalently sparsify the interaction matrix. One possibility is to use a convolutional structure so that each token only interacts with a few neighboring tokens. Across multiple layers, tokens still interact at larger distances as the receptive field expands. As for convolution in images, the kernel can vary in size and dilation rate.

A pure convolutional approach might require many layers to integrate information over large distances. One way to speed this process up is to allow select tokens (perhaps at the start of every sentence) to attend to all the other tokens (encoder model) or all the previous tokens (decoder model). A similar idea is to have a small number of global tokens that connect to all the other tokens and themselves. Like the <cls> token, these do not actually represent any word, but simply serve to provide connections between distant parts of the input sequence.

12.10 Transformers for images

Transformers were initially developed for text data. Their enormous success in this area led to experimentation on images. This was not obviously a promising idea for two reasons. First, there are many more pixels in an image than words in a sentence, and so the quadratic complexity of self-attention poses a practical bottleneck. Second, convolutional nets are designed to have a good inductive bias because each layer is equivariant to spatial translation. However, this must be learned in a transformer network.

Regardless of these apparent disadvantages, transformer networks for images have now eclipsed the performance of convolutional networks for image classification and other tasks. This is partly because of the enormous scale at which they can be built, and the large amounts of data that can be used to pre-train the networks. In this section, we'll describe several applications of the transformer to modeling images.

12.10.1 ImageGPT

ImageGPT is a transformer decoder; it builds an autoregressive model of image pixels that ingests a partial image and predicts the subsequent pixel value. The quadratic complexity of the transformer network means that the largest model (which contained 6.8 billion parameters) could still only operate on 64×64 images. Moreover, to make this tractable, the original 24-bit RGB color space had to be quantized into a nine-bit color space, so the system ingests (and predicts) one of 512 possible tokens at each position.

Images are naturally 2D objects, but ImageGPT simply learns a different position embedding at each pixel. Hence it must learn that each pixel not only has a close relationship with its preceding neighbors, but also with nearby pixels in the row above. Figure 12.15 shows example generation results.

The internal representation of this decoder was also used as a basis for image classification. The final representations for each pixel are averaged and a linear layer maps to activations which are subsequently passed through a softmax layer to predict class probabilities. The system is pre-trained on a large corpus of web images and then is fine-tuned on the ImageNet database resized to 48×48 pixels using a loss function that contains both a cross-entropy term for image classification and a generative loss term for predicting the pixels. Despite using a large amount of external training data, the system achieved only a 27.4% top-1 error rate on ImageNet (figure 10.15). This was less than convolutional architectures of the time (see figure 10.22) but is still impressive given the very small input image size; it is unsurprising that it fails to classify images where the object is small or thin.

12.10.2 Vision Transformer (ViT)

The *Vision Transformer* tackled the problem of image resolution by dividing the image into 16×16 patches (figure 12.16). Each of these is mapped to a lower dimension via a learned linear transformation and it is these representations that are fed into the

Problem 12.8

Figure 12.15 ImageGPT. a) Images generated from the autoregressive ImageGPT model. The top left pixel is chosen from the marginal probability distribution. Subsequent pixels are generated in turn, working along the rows until the bottom left of the image is reached. For each pixel, the transformer decoder generates a conditional distribution as in equation 12.14, and a sample is drawn. The extended sequence is then fed back into the network to generate the next pixel, and so on. b) Image completion. In each case, the lower half of the image is removed (top row) and ImageGPT completes the remaining part pixel by pixel (three different completions shown). Adapted from https://openai.com/blog/image-gpt/.

transformer network. Once again, standard 1D position embeddings were learned.

This model has an encoder architecture with a <cls> token (see figures 12.9–12.10). However, unlike BERT it used *supervised* pre-training on a large labeled database of 303 million labeled images from 18,000 classes. The <cls> token was mapped via a final neural network layer to create activations that are fed into a softmax function to generate class probabilities. After pre-training, the system is applied to the final classification task by replacing this final layer with one that maps to the desired number of classes and is then fine-tuned.

For the ImageNet benchmark, this system achieved a remarkable 11.45% top-1 error rate. It is notable, however, that it did not perform as well as the best contemporary convolutional networks without supervised pre-training. It appears that the strong inductive bias of convolutional networks can only be superseded by employing extremely large amounts of training data.

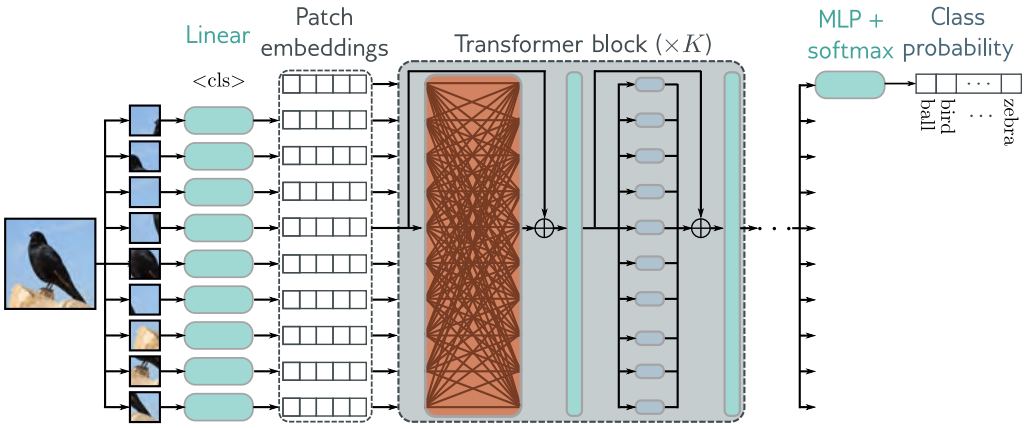

Figure 12.16 Vision transformer. The Vision Transformer (ViT) breaks the image into a grid of patches (16×16 in the original implementation) and each of these is projected via a learned linear transformation to become a patch embedding. These patch embeddings are fed into a transformer encoder network and the <cls> token is used to predict the class probabilities.

12.10.3 Multi-scale vision transformers

The Vision Transformer differs from convolutional architectures in that it operates on a single scale. Many transformer models that process the image at multiple scales have subsequently been proposed. Similarly to convolutional networks, these generally start with high-resolution patches and few channels and gradually decrease the resolution, while simultaneously increasing the number of channels.

A representative example of a multi-scale transformer is the *shifted-window* or *SWIN* transformer. This is an encoder transformer that divides the image into patches, and groups these patches into a grid of windows within which self-attention is applied independently (figure 12.17). These windows are shifted in adjacent transformer blocks, so the effective receptive field at a given patch can expand beyond the window border.

The scale is reduced periodically by concatenating features from non-overlapping 2×2 patches and applying a linear transformation that maps these concatenated features to twice the original number of channels. This architecture does not have a <cls> token but instead averages the output features at the last layer. These are then mapped via a linear layer to the desired number of classes and passed through a softmax function to output class probabilities. At the time of writing, the most sophisticated version of this architecture achieves a 9.89% top-1 error rate on the ImageNet database.

A related idea is to periodically integrate information from across the whole image. *Dual attention vision transformers* (DaVIT) achieve this by alternating two types of transformer blocks. In the first, different patches of the image attend to one another, and the self-attention computation uses all the channels. In the second, the different

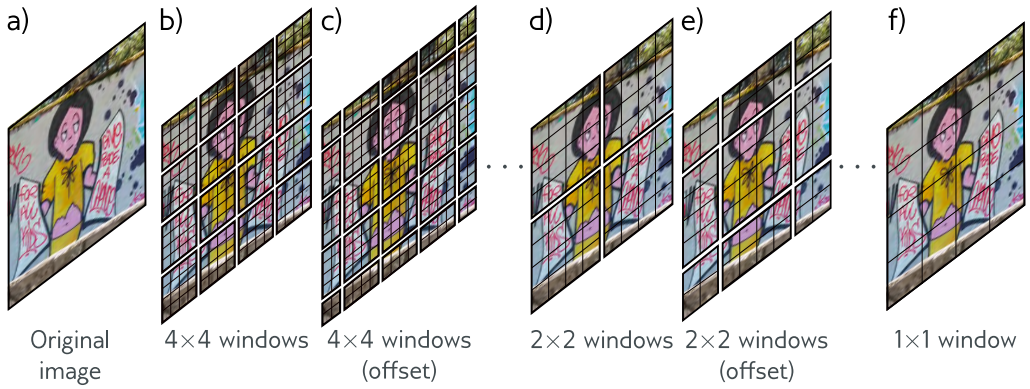

Figure 12.17 Shifted window (SWIN) transformer (Liu et al. 2021d). a) Original image. b) The SWIN transformer breaks the image into a grid of windows and each of these windows into a sub-grid of patches. The transformer network applies self-attention to the patches within each window independently. c) Each alternate layer shifts the windows so that the subsets of patches that interact with one another change and information can propagate across the whole image. d) After several layers, the 2×2 blocks of patch representations are concatenated so the effective patch (and window) size increase. e) Alternate layers use shifted windows at this new lower resolution. f) Eventually, the resolution is such that there is just a single window and the patches span the entire image.

channels attend to one another, and the self-attention computation uses all of the spatial positions. This architecture reaches a 9.60% top-1 error rate on ImageNet and is close to the state-of-the-art at the time of writing.

Problem 12.9

12.11 Summary

This chapter introduced self-attention and then described how this forms part of the transformer architecture. We then presented the encoder, decoder, and encoder-decoder versions of this architecture. We've seen that the transformer operates on sets of high-dimensional embeddings. It has a low computational complexity per layer and much of the computation can be performed in parallel, using the matrix form. Since every input embedding interacts with every other, it can describe long-range dependencies in text. Ultimately though, the computation scales quadratically with the sequence length; one approach to reducing the complexity is to sparsify the interaction matrix.

One of the advantages of transformers is that they can be trained with extremely large unlabeled datasets. This is the first example of *unsupervised learning* (learning without labels) that we have seen in this book. The encoder model trains a text representation that can be used for other tasks by predicting missing tokens. The decoder model goes

one step further and builds an autoregressive probability model over the input tokens. This is the first example of a *generative model* that we have seen in this book; sampling from generative models creates new data examples and examples of text and image generation were provided.

In the next chapter, we consider networks that are used to process graph data. These have close connections with transformers in that they involve a series of network layers in which the nodes of the graph interact with each other. In chapters 16–17, we return to generative models.

Notes

Natural language processing: Transformers were developed for natural language processing (NLP) tasks. This is an enormous area that deals with text analysis, categorization, generation, and manipulation. Example tasks include part of speech tagging, translation, text classification, entity recognition (people, places, companies, etc.), text summarization, question answering, word sense disambiguation, and document clustering. NLP was originally tackled by rule-based methods that exploited the structure and statistics of grammar (see Manning & Schutze 1999 and Jurafsky & Martin 2000 for early approaches).

Recurrent neural networks: Prior to the introduction of transformers, many state-of-the-art NLP applications used *recurrent neural networks*, or *RNNs* for short (figure 12.18). The term "recurrent" was introduced by Rumelhart et al. (1985), but the main idea dates to at least Minsky & Papert (1969). RNNs ingest a sequence of inputs (words in NLP) one at a time. At each step, the network receives both the new input and a hidden representation computed from the previous time step (the recurrent connection). The final output contains information about the whole input. This representation can then be used to support NLP tasks like classification or translation. They have also been used in a decoding context in which generated tokens are fed back into the model to form the next input to the sequence. For example, the PixelRNN (Van Oord et al. 2016) used RNNs to build an autoregressive model of images.

From RNNs to transformers: One of the problems with RNNs is that they can forget information that is further back in the sequence. More sophisticated versions of this architecture such as long short-term memory networks or LSTMs (Hochreiter & Schmidhuber 1997) and gated recurrent units or GRUs (Cho et al. 2014; Chung et al. 2014) partially addressed this problem. However, in machine translation the idea emerged that all of the intermediate representations could be used as a basis for producing the output sentence. Moreover, certain output words should *attend* more to certain input words, according to their relation (Bahdanau et al. 2014). This ultimately led to dispensing with the recurrent structure altogether and replacing it with the encoder-decoder transformer (Vaswani et al. 2017). Here input tokens attend to one another (self-attention), output tokens attend to those earlier in the sequence (masked self-attention), and output tokens also attend to the input tokens (cross-attention). A formal algorithmic description of the transformer can be found in Phuong & Hutter (2022), and a survey of work can be found in Lin et al. (2021). The literature should be approached with caution, as many enhancements to transformers do not make meaningful performance improvements when carefully assessed in controlled experiments (Narang et al. 2021).

Applications: Models based on self-attention and/or the transformer architecture have been applied to text sequences (Vaswani et al. 2017), image patches (Dosovitskiy et al. 2020), protein sequences (Rives et al. 2021), graphs (Veličković et al. 2017), database schema (Xu et al.

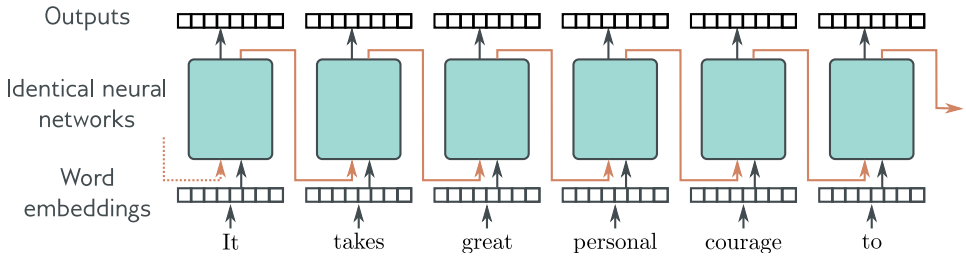

Figure 12.18 Recurrent neural networks. The word embeddings are passed sequentially through a series of identical neural networks. Each network has two outputs; one is the output embedding and the other (orange arrows) feeds back into the next neural network, along with the next word embedding. Each output embedding contains information about the word itself and its context in the preceding sentence fragment. In principle, the final output contains information about the entire sentence and could be used to support classification tasks in a similar way to the <cls> token in a transformer encoder model. However, the network sometimes gradually forgets about tokens that are further back in time.

2021b), speech (Wang et al. 2020c), mathematical integration (formulated as a translation problem) (Lample & Charton 2020), and time series (Wu et al. 2020b). However, their most celebrated successes have been in building language models, and more recently as a replacement for convolutional networks in computer vision.

Language models: The work of Vaswani et al. (2017) targeted translation tasks, but transformers are now more usually used to build either pure encoder or pure decoder models, the most famous of which are BERT (Devlin et al. 2018) and GPT2/GPT3 (Radford et al. 2019; Brown et al. 2020), respectively. These models are usually tested against benchmarks like GLUE (Wang et al. 2018a), which includes the SQuAD question-answering task (Rajpurkar et al. 2016) described in section 12.6.2, SuperGLUE (Wang et al. 2019) and BIG-bench (forthcoming 2022), which combine many NLP tasks to create an aggregate score for measuring language ability. Decoder models are generally not fine-tuned for these tasks but can perform well anyway when given a few examples of questions and answers and asked to complete the text from the next question. This is referred to as *few shot learning* (Brown et al. 2020).

Since GPT3, many decoder language models have been released with steady improvement in few-shot results. These include GLaM (Du et al. 2022), Gopher (Rae et al. 2021), Chinchilla (Hoffmann et al. 2022), Megatron-Turing NLG (Smith et al. 2022), and LaMDa (Thoppilan et al. 2022). Most of the performance improvement is attributable to increased model size, using sparsely activated modules, and exploiting larger datasets. At the time of writing, the most recent model is PaLM (Chowdhery et al. 2022), which has 540 billion parameters and was trained on 780 billion tokens across 6144 processors. It is interesting to note that since text is highly compressible, this model has more than enough capacity to memorize the entire training dataset. This is true for many language models. Many bold statements have been made about how large language models exceed human performance. This is probably true for some tasks, but such statements should be treated with caution (see Ribeiro et al. 2020, McCoy et al. 2019, Bowman & Dahl 2021, and Dehghani et al. 2021).

These models have considerable knowledge about the world. For example, in the example in section 12.7.4, the model knows key facts about Deep Learning, including that it is a type

of Machine Learning, and that it has algorithms and applications. Indeed, one such model has been mistakenly identified as being sentient (Clark 2022). However, there are persuasive arguments that the degree of "understanding" that this type of model can ever have is limited (Bender & Koller 2020).

Tokenizers: Schuster & Nakajima (2012) and Sennrich et al. (2015) introduced *WordPiece* and *byte pair encoding* (*BPE*) respectively. Both of these methods greedily merge pairs of tokens based on their frequency of adjacency, with the main difference being in how the initial tokens are chosen. For example, in BPE, the initial tokens are characters or punctuation with a special token to denote whitespace. The merges cannot occur over the whitespace. As the algorithm proceeds, new tokens are formed by combining characters recursively so that sub-word and word tokens emerge. The unigram language model (Kudo 2018) generates several possible candidate merges and chooses the best one based on the likelihood in a language model. Provilkov et al. (2019) develop BPE dropout, which generates the candidates more efficiently by introducing randomness into the process of counting frequencies. Versions of both byte pair encoding and the unigram language model are included in the SentencePiece library (Kudo & Richardson 2018), which works directly on Unicode characters, and so can work with any language. He et al. (2020b) introduce a method that treats the sub-word segmentation as a latent variable that should be marginalized out for learning and inference.

Decoding algorithms: Transformer decoder models take a body of text and return a probability over the next token. This is then added to the preceding text and the model is run again. The process of choosing tokens from these probability distributions is known as *decoding*. Näive ways to do this would just be to either (i) greedily choose the most likely token or (ii) choose a token at random according to the distribution. However, neither of these methods works well in practice. In the former case, the results may be very generic, and the latter case may lead to degraded quality outputs (Holtzman et al. 2019). This is partly because during training the model was only exposed to sequences of ground truth tokens (known as *teacher forcing*) but sees its own output when deployed.

It is not computationally feasible to try every combination of tokens in the output sequence, but it is possible to maintain a fixed number of parallel hypotheses and choose the most likely overall sequence. This is known as *beam search*. Beam search tends to produce many similar hypotheses and has been modified to investigate more diverse sequences (Vijayakumar et al. 2016; Kulikov et al. 2018). One possible problem with random sampling is that there is a very long tail of unlikely following words. This has led to the development of *top-K sampling*, in which tokens are sampled from only the K most likely hypotheses (Fan et al. 2018). This still sometimes allows unreasonable token choices when there are only a few high probability choices. To resolve this problem Holtzman et al. (2019) proposed *nucleus sampling*, in which tokens are sampled from a fixed proportion of the total probability mass. These issues are discussed in more depth by El Asri & Prince (2020).

Types of attention: Scaled dot-product attention (Vaswani et al. 2017) is just one of a family of attention mechanisms that includes additive attention (Bahdanau et al. 2014), multiplicative attention (Luong et al. 2015), key-value attention (Daniluk et al. 2017), and memory compressed attention (Liu et al. 2018c). Other work (Zhai et al. 2021) has constructed "attention-free" transformers, in which the tokens interact in a way that does not have quadratic complexity. Multi-head attention was also introduced by Vaswani et al. (2017). Interestingly, it appears that most of the heads can be pruned after training without critically affecting the performance (Voita et al. 2019); it has been suggested that their role is to guard against bad initializations.

Relationship of self-attention to other models: The self-attention computation has close connections to other models. First, it is a case of a hypernetwork (Ha et al. 2016) in that it uses one part of the network to choose the weights of another part: the attention matrix forms

the weights of a sparse network layer that maps the values to the outputs (figure 12.3). The *synthesizer* (Tay et al. 2021) simplifies this idea by simply using a neural network to create each row of the attention matrix from the corresponding input. Even though the input tokens no longer interact with each other, this works surprisingly well. Wu et al. (2019) present a similar system that produces an attention matrix with a convolutional structure, so the tokens attend to their neighbors. The gated multi-layer perceptron (Wu et al. 2019) computes a matrix that pointwise multiplies the values, and so modifies them without mixing them. Transformers are also closely related to *fast weight memory systems*, which were the intellectual forerunners of hypernetworks (Schlag et al. 2021).

Self-attention can also be thought of as a routing mechanism (figure 12.1) and from this viewpoint there is a connection to capsule networks (Sabour et al. 2017). These capture hierarchical relations in images so lower network levels might detect facial parts (noses, mouths), which are then combined (routed) in higher level capsules that represent a face. However, capsule networks use *routing by agreement*. In self-attention, the inputs compete with each other for how much they contribute to a given output (via the softmax operation). In capsule networks, the outputs of the layer compete with each other for inputs from lower levels. Once we consider self-attention as a routing network, we can question whether it is necessary to make this routing dynamic (i.e., dependent on the data). The random synthesizer (Tay et al. 2021) removed the dependence of the attention matrix on the inputs entirely and either used predetermined random values or learned values. This performed surprisingly well across a variety of tasks.

Multi-head self-attention also has close connections to graph neural networks (see chapter 13), convolution (Cordonnier et al. 2019), recurrent neural networks (Choromanski et al. 2020), and memory retrieval in Hopfield networks (Ramsauer et al. 2020). For more information on the relation between transformers and other models, consult Prince (2021a).

Position encoding: The original transformer paper (Vaswani et al. 2017) experimented with pre-defining the position embedding matrix $\boldsymbol{\Pi}$, and learning the position embedding $\boldsymbol{\Pi}$. It might seem odd to *add* the position embeddings to the $D \times N$ data matrix $\mathbf{X}$ rather than concatenate them. However, since the data dimension D is usually much greater than the number of tokens N, the position embedding lies in a subspace. The word embeddings in $\mathbf{X}$ are learned, and so it's possible in theory for the system to keep the two components in orthogonal subspaces and retrieve the position embeddings as required. The predefined embeddings chosen by Vaswani et al. (2017) were a family of sinusoidal components that had two attractive properties: (i) the relative position of two embeddings is easy to recover using a linear operation and (ii) their dot product generally decreased as the distance between positions increased (see Prince 2021a for more details). Many systems such as GPT3 and BERT used learned embedding matrices. Wang et al. (2020a) examined the cosine similarities of the learned position embeddings in these models and showed that they generally decline with relative distance, although they also have a periodic component.

Much subsequent work has modified just the attention matrix, so that in the scaled dot product self-attention equation:

$$\mathbf{Sa}[\mathbf{X}] = \mathbf{V} \cdot \mathbf{Softmax}\left[\frac{\mathbf{K}^T\mathbf{Q}}{\sqrt{D_q}}\right] \tag{12.16}$$

only the keys and values contain position information:

$$\begin{aligned}\mathbf{V} &= \boldsymbol{\beta}_v\mathbf{1}^\mathbf{T} + \boldsymbol{\Omega}_\mathbf{v}\mathbf{X} \\ \mathbf{Q} &= \boldsymbol{\beta}_q\mathbf{1}^\mathbf{T} + \boldsymbol{\Omega}_\mathbf{q}(\mathbf{X}+\boldsymbol{\Pi}) \\ \mathbf{K} &= \boldsymbol{\beta}_k\mathbf{1}^\mathbf{T} + \boldsymbol{\Omega}_\mathbf{k}(\mathbf{X}+\boldsymbol{\Pi}).\end{aligned} \tag{12.17}$$

This has led to the idea of multiplying out the quadratic term in the numerator of equation 12.16 and retaining only some of the terms. For example, Ke et al. (2020) decouple or *untie* the content and position information by retaining only the content-content and position-position terms and using different projection matrices $\boldsymbol{\Omega}_\bullet$ for each.

Another modification is to directly inject information about the relative position. This is more important than absolute position since a batch of text can start at an arbitrary place in a document. Shaw et al. (2018), Raffel et al. (2020), and Huang et al. (2020b) all developed systems where a single term was learned for each relative position offset, and the attention matrix was modified in various ways using these *relative position embeddings*. Wei et al. (2019) investigated relative position embeddings based on predefined sinusoidal embeddings rather than learned values. DeBERTa (He et al. 2020a) combines all of these ideas; they retain only a subset of terms from the quadratic expansion, apply different projection matrices to them, and use relative position embeddings. Other work has explored sinusoidal embeddings that encode absolute and relative position information in more complex ways (Su et al. 2021).

Wang et al. (2020a) empirically compare the performance of transformers in BERT with different position embeddings. They found that relative position embeddings perform better than absolute position embeddings, but that there was not much difference between using sinusoidal and learned embeddings. A survey of position embeddings for transformers can be found in Dufter et al. (2021).

Extending transformers to longer sequences: The complexity of the self-attention mechanism increases quadratically with the sequence length. Some tasks like summarization or question answering may require long inputs, and so this quadratic dependence limits performance. Three lines of work have attempted to address this problem. The first decreases the size of the attention matrix, the second makes the attention sparse, and the third modifies the attention mechanism to make it more efficient.

To decrease the size of the attention matrix, Liu et al. (2018b) introduced *memory compressed attention*. This applies strided convolution to the keys and values, which reduces the number of positions in a very similar way to a downsampling operation in a convolutional network. One way to think about this is that attention is applied between weighted combinations of neighboring positions, where the weights are learned. Along similar lines, Wang et al. (2020b) observed that the quantities in the attention mechanism are often low rank in practice and developed the *LinFormer*, which projects the keys and values onto a smaller subspace before computing the attention matrix.

To make attention sparse, Liu et al. (2018b) proposed *local attention*, in which neighboring blocks of tokens only attend to one another. This creates a block diagonal interaction matrix (see figure 12.14). Obviously, this means that information cannot pass from block to block, and so such layers are typically alternated with full attention. Along the same lines, GPT3 (Brown et al. 2020) uses a convolutional interaction matrix and alternates this with full attention. Child et al. (2019) and Beltagy et al. (2020) experimented with various interaction matrices, including using convolutional structures with different dilation rates but allowing some queries to interact with every other key. Ainslie et al. (2020) introduced the *extended transformer construction* (figure 12.14h), which uses a set of global embeddings that interact with every other token. This can only be done in the encoder version, or these implicitly allow the system to "look ahead". When combined with relative position encoding, this scheme requires special encodings for mapping to, from, and between these global embeddings. *BigBird* Ainslie et al. (2020) combined global embeddings and a convolutional structure with a random sampling of possible connections. Other work has investigated learning the sparsity pattern of the attention matrix (Roy et al. 2021; Kitaev et al. 2020; Tay et al. 2020a).

Finally, it has been noted that the terms in the numerator and denominator of the softmax operation that computes attention have the form $\exp[\mathbf{k}^T\mathbf{q}]$. This can be treated as a kernel function and as such can be expressed as the dot product $\mathbf{g}[\mathbf{k}]^T\mathbf{g}[\mathbf{q}]$ where $\mathbf{g}[\bullet]$ is a nonlinear

Problem 12.10

transformation. This formulation decouples the queries and keys and makes the attention computation more efficient. Unfortunately, to replicate the form of the exponential terms, the transformation $\mathbf{g}[\bullet]$ must map the inputs to the infinite space. The linear transformer (Katharopoulos et al. 2020) recognizes this and replaces the exponential term with a different measure of similarity. The *Performer* (Choromanski et al. 2020) approximates this infinite mapping with a finite-dimensional one.

More details about extending transformers to longer sequences can be found in Tay et al. (2020b) and Prince (2021a).

Training transformers: Training transformers is challenging and requires both learning rate warm-up (Goyal et al. 2018) and Adam (Kingma & Ba 2014). Indeed Xiong et al. (2020a) and Huang et al. (2020a) show experimentally that gradients vanish, and the Adam updates decrease in magnitude without learning rate warm-up. Several interacting factors cause these problems. The residual connections cause the gradients to explode (figure 11.6) and normalization layers are required to prevent this. Vaswani et al. (2017) used LayerNorm rather than BatchNorm because NLP statistics are highly variable between batches, although subsequent work has modified BatchNorm for transformers (Shen et al. 2020). The positioning of the LayerNorm outside of the residual block causes gradients to shrink as they pass back through the network (Xiong et al. 2020a). In addition, the relative weight of the residual connections and main self-attention mechanism varies as we move through the network upon initialization (see figure 11.6c), and there is the additional complication that the gradients for the query and key parameter are much smaller than for the value parameters (Liu et al. 2020), and this necessitates the use of Adam. These factors interact in a complex way and make training unstable, necessitating the use of learning rate warm-up.

There have been various attempts to make training more stable including (i) a variation of FixUp called *TFixup* (Huang et al. 2020a) that allows the LayerNorm components to be removed, (ii) moving the LayerNorm components in the network (Liu et al. 2020), and (iii) reweighting the two paths in the residual branches (Liu et al. 2020; Bachlechner et al. 2021). Xu et al. (2021b) introduce an initialization scheme called *DTFixup* that allows transformers to be trained with smaller datasets. A detailed discussion of these issues can be found in Prince (2021b).

Applications in vision: ImageGPT (Chen et al. 2020a) and the Vision Transformer (Dosovitskiy et al. 2020) were both early transformer architectures applied to images. Transformers have been used for image classification (Dosovitskiy et al. 2020; Touvron et al. 2021), object detection (Carion et al. 2020; Zhu et al. 2020; Fang et al. 2021), semantic segmentation (Ye et al. 2019; Xie et al. 2021; Gu et al. 2022), super-resolution (Yang et al. 2020a), action recognition (Sun et al. 2019; Girdhar et al. 2019), image generation (Chen et al. 2021), visual question answering (Su et al. 2019b; Tan & Bansal 2019), inpainting (Wan et al. 2021; Zheng et al. 2021; Zhao et al. 2020b; Li et al. 2022), colorization (Kumar et al. 2021) and many other vision tasks. Surveys of transformers for vision can be found in Khan et al. (2021) and Liu et al. (2021c).

Transformers and convolutional networks: Transformers have been combined with convolutional neural networks to solve diverse computer vision tasks including image classification (Wu et al. 2020a), object detection (Hu et al. 2018; Carion et al. 2020), video processing (Wang et al. 2018b; Sun et al. 2019), unsupervised object discovery (Locatello et al. 2020) and various text/vision tasks (Chen et al. 2020d; Lu et al. 2019; Li et al. 2019). Transformers can outperform convolutional networks for vision tasks but usually require large quantities of data to achieve superior performance. Often, they are pre-trained on the enormous JFT dataset (Sun et al. 2017). The transformer does not have the inductive bias of convolutional networks but it appears that by using gargantuan amounts of data they can surmount this disadvantage.

From pixels to video: Non-local networks (Wang et al. 2018b) were an early application of self-attention to image data. Transformers themselves were initially applied to pixels in local

neighborhoods (Parmar et al. 2018; Hu et al. 2019; Ramachandran et al. 2019; Zhao et al. 2020a). ImageGPT (Chen et al. 2020a) scaled this to model all of the pixels in a (small) image. The Vision Transformer (ViT) (Dosovitskiy et al. 2020) used non-overlapping patches to analyze bigger images.

Since then, many multi-scale systems have been developed including the SWIN transformer (Liu et al. 2021d), SWINV2 (Liu et al. 2022), multi-scale transformers (MVIT) (Fan et al. 2021), and pyramid vision transformers (Wang et al. 2021a). The Crossformer (Wang et al. 2021b) models interactions between spatial scales. Ali et al. (2021) introduced cross-covariance image transformers, in which the channels rather than spatial positions attend to one another, hence making the size of the attention matrix indifferent to the image size. The dual attention vision transformer was developed by Ding et al. (2022) and alternates between local spatial attention within sub-windows and spatially global attention between channels. Chu et al. (2021) similarly alternate between local attention within sub-windows and global attention by subsampling the spatial domain. Dong et al. (2022) adapt the ideas of figure 12.14, in which the interactions between elements are sparsified to the 2D image domain.

Transformers were subsequently adapted to video processing (Arnab et al. 2021; Bertasius et al. 2021; Liu et al. 2021d; Neimark et al. 2021; Patrick et al. 2021). A survey of transformers applied to video can be found in Selva et al. (2022).

Combining image and text: CLIP (Radford et al. 2021) learns a joint encoder for images and their captions using a contrastive pre-training task. The system ingests N images and their captions and produces a matrix of compatibility between images and captions. The loss function encourages the correct pairs to have a high score and the incorrect pairs to have a low score. Ramesh et al. (2021) and Ramesh et al. (2022) train a diffusion decoder to invert the CLIP image encoder (DALL-E) for text-conditional image generation (see chapter 17).

Problems

Problem 12.1 Consider a self-attention mechanism that processes N inputs of length D to produce N outputs of the same size. How many weights and biases are used to compute values? How many attention weights $\mathrm{a}[\bullet,\bullet]$ will there be? How many weights and biases would there be if we build a fully connected network relating all DN inputs to all DN outputs?

Problem 12.2 Why might we want to ensure that the input to the self-attention mechanism is the same size as the output?

Problem 12.3 The self-attention mechanism is defined as:

$$\mathbf{X} = (\boldsymbol{\beta}_v\mathbf{1}^\mathbf{T} + \boldsymbol{\Omega}_\mathbf{v}\mathbf{X})^\mathbf{T}\mathbf{Softmax}[(\boldsymbol{\beta}_\mathbf{q}\mathbf{1}^\mathbf{T} + \boldsymbol{\Omega}_\mathbf{q}\mathbf{X})^\mathbf{T}(\boldsymbol{\beta}_\mathbf{v}\mathbf{1}^\mathbf{T} + \boldsymbol{\Omega}_\mathbf{k}\mathbf{X})]. \tag{12.18}$$

Find the derivatives of the output $\mathbf{X}$ with respect to the parameters $\boldsymbol{\beta}_v, \boldsymbol{\Omega}_v, \boldsymbol{\beta}_q, \boldsymbol{\Omega}_q, \boldsymbol{\beta}_k$ and $\boldsymbol{\Omega}_k$.

Problem 12.4 Consider the softmax operation:

$$y_k = \mathrm{softmax}_k[\mathbf{z}] = \frac{\exp[z_k]}{\sum_{k=1}^{K}\exp[z_k]}, \tag{12.19}$$

in the case where there are 5 inputs with values: $z_1 = -3$, $z_2 = 1$, $z_3 = 1000$, $z_4 = 5$, $z_5 = -1$. Compute the 25 derivatives, $\partial y_k[\mathbf{z}]/\partial z_j$ for all $j, k \in \{1, 2, 3, 4, 5\}$. What do you conclude?

Problem 12.5 Why is implementation more efficient if the values, queries, and keys in each of the H heads each have dimension D/H where D is the original dimension of the data?

Problem 12.6 Write Python code to create sub-word tokens for the nursery rhyme in figure 12.7 using byte-pair encoding.

Problem 12.7 BERT was pre-trained using two tasks. The first task requires the system to predict missing (masked) words. The second task requires the system to classify pairs of sentences as being adjacent or not in the original text. Identify whether each of these tasks is generative or contrastive (see section 9.3.6). Why do you think they used two tasks? Propose two novel contrastive tasks that could be used to pre-train a language model.

Problem 12.8 One problem with vision transformers is that the computation expands quadratically with the number of patches. Devise two methods to reduce the amount of computation using the principles from figure 12.14.

Problem 12.9 Consider representing an image with a grid of 16×16 patches, each of which is represented by a patch embedding of length 512. Compare the amount of computation required in the DaViT transformer to perform attention (i) between the patches, using all of the channels, and (ii) between the channels, using all of the patches.

Problem 12.10 Attention values are normally computed as:

$$\begin{aligned} a[\mathbf{x}_n, \mathbf{x}_m] &= \text{softmax}_m\left[\mathbf{q}_m^T\mathbf{k}_n\right] \\ &= \frac{\exp\left[\mathbf{q}_m^T\mathbf{k}_n\right]}{\sum_{m'=1}^{N}\exp\left[\mathbf{q}_{m'}^T\mathbf{k}_n)\right]}. \end{aligned} \tag{12.20}$$

Consider replacing $\exp\left[\mathbf{q}_m^T\mathbf{k}_n\right]$ with the dot product $\mathbf{g}[\mathbf{q}_m]^T\mathbf{g}[\mathbf{k}_n]$ where $\mathbf{g}[\bullet]$ is a nonlinear transformation. Show how this makes the computation of the attention values more efficient.

Chapter 13

Graph neural networks

Chapter 10 described convolutional networks, which specialize in processing regular arrays of data (e.g., images). Chapter 12 described transformers, which specialize in processing sequences of variable length (e.g., text). This chapter describes *graph neural networks*. As the name suggests, these are neural architectures that process graphs (i.e., sets of nodes connected by edges).

There are three novel challenges associated with processing graphs. First, their topology is variable, and it is hard to design networks that are both expressive and can cope with this variation. Second, graphs may be enormous; a graph representing connections between users of a social network might have a billion nodes. Third, there may only be a single monolithic graph available, so the usual protocol of training with many data examples and testing with new data may not be appropriate.

This chapter starts by providing some examples of graphs and then describes how they are encoded and common problem formulations. The algorithmic requirements for processing graphs are discussed and this leads naturally to *graph convolutional networks*, which are a particular type of graph neural network.

13.1 What is a graph?

A graph is a very general structure and consists of a set of *nodes* or *vertices*, where pairs of nodes are connected by *edges* or *links*. Graphs are typically sparse; only a small subset of the possible edges are present.

Some objects in the real world naturally take the form of graphs. For example, road networks can be considered graphs in which the nodes are physical locations and the edges represent roads between them (figure 13.1a). Chemical molecules are small graphs in which the nodes represent atoms and the edges represent chemical bonds (figure 13.1b). Electrical circuits are graphs in which the nodes represent components and junctions, and the edges are electrical connections (figure 13.1c).

Furthermore, many datasets can also be represented by graphs, even if this is not their obvious surface form. For example:

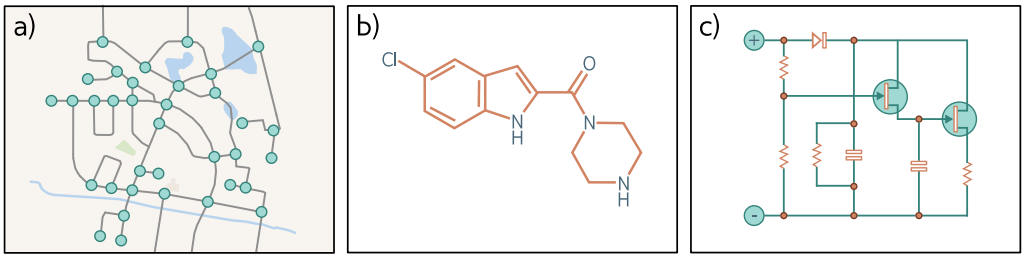

Figure 13.1 Real-world graphs. Some objects such as a) road networks, b) molecules, and c) electrical circuits are naturally structured as graphs.

- Social networks are graphs where nodes are people and the edges represent friendships between them.
- The scientific literature can be viewed as a graph in which the nodes are papers and the edges represent citations.
- Wikipedia can be thought of as a graph in which the nodes are articles and the edges represent hyperlinks between articles.
- Computer programs can be represented as graphs in which the nodes are syntax tokens (variables at different points in the program flow) and the edges represent computations involving these variables.
- Geometric point clouds can be represented as graphs. Here, each point is a node and has edges that connect to other nearby points.
- Protein interactions in a cell can be expressed as graphs, where the nodes are the proteins, and there is an edge between two proteins if they interact.

In addition, a set (an unordered list) can be considered a special case of a graph, where every member is a node and connects to every other. Similarly, an image can be treated as a graph with regular topology, in which each pixel is a node and has edges to its eight adjacent neighbors.

13.1.1 Types of graph

Graphs are very general structures and can be categorized in various ways. The social network in figure 13.2a contains *undirected edges*; each pair of individuals with a connection between them have mutually agreed to be friends, and so there is no sense that the relationship is directional. In contrast, the citation network in figure 13.2b contains *directed edges*. Each paper cites other papers, and this relationship is inherently one-way.

Figure 13.2c depicts a *knowledge graph* that encodes a set of facts about objects by defining relations between them. Technically, this is a *directed heterogeneous multigraph*. It is heterogeneous because the nodes can represent different types of entities (e.g., people,

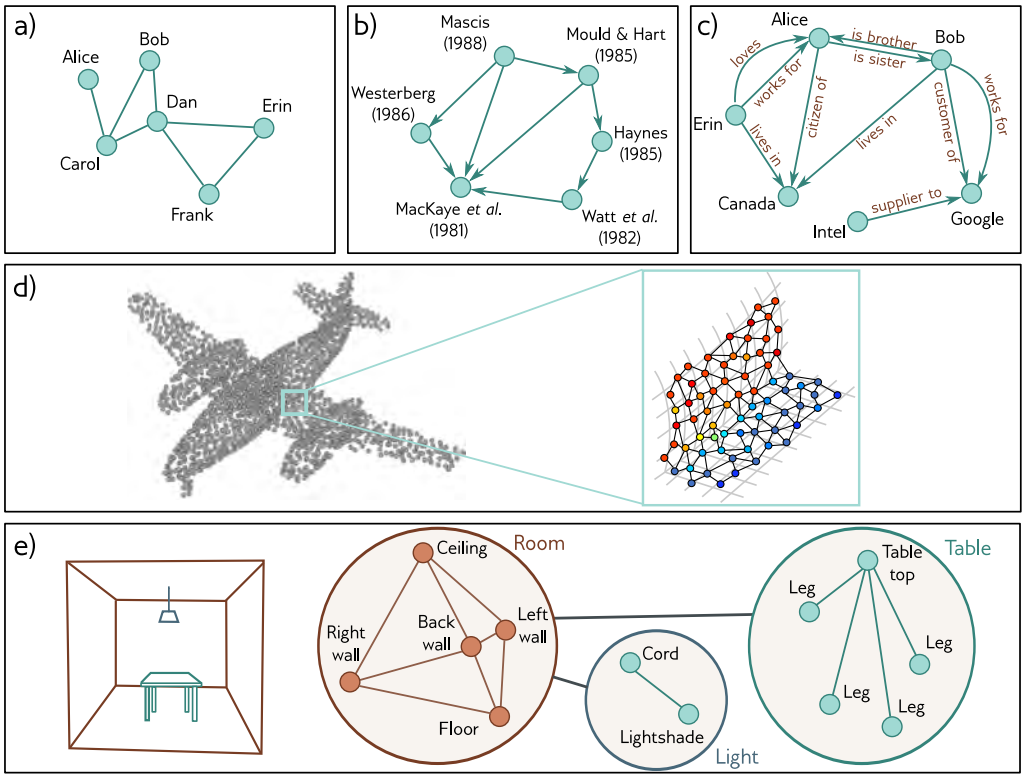

Figure 13.2 Types of graph. a) A social network is an undirected graph; the connections between people are symmetric. b) A citation network is a directed graph; one publication cites another, so the relationship is asymmetric. c) A knowledge graph is a directed heterogeneous multigraph. The nodes are heterogeneous in that they represent different object types (people, places, companies) and there may be multiple edges representing different relations between each node. d) A point set can be converted to a graph by forming edges between nearby points. Each node has an associated position in 3D space and this is termed a geometric graph (adapted from Hu et al. 2021). e) The scene on the left can be represented by a hierarchical graph. The topology of the room, table, and light are all represented by graphs and these graphs form nodes in a larger graph representing object adjacency (adapted from Fernández-Madrigal & González 2002).

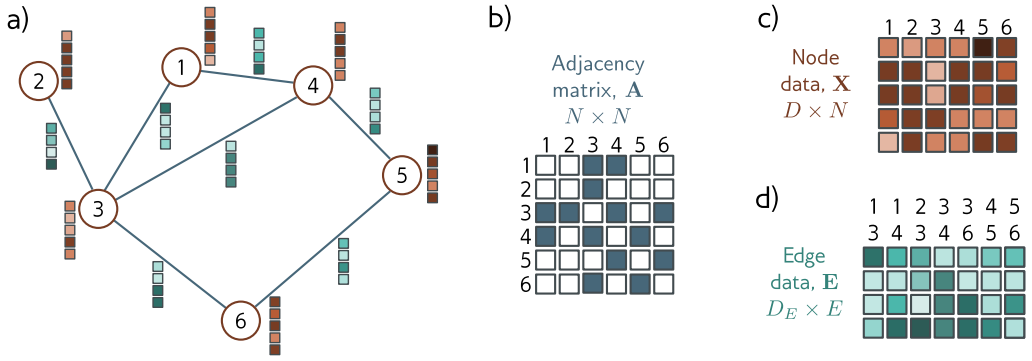

Figure 13.3 Graph representation. a) Example graph with six nodes and seven edges. Each node has an associated embedding of length five (brown vectors). Each edge has an associated embedding of length four (blue vectors). This graph can be represented by three matrices. b) The adjacency matrix is a binary matrix, where the element (m, n) is set to one if node m connects to node n. c) The node data matrix $\mathbf{X}$ contains the concatenated node embeddings. d) The edge data matrix $\mathbf{E}$ contains the edge embeddings.

countries, companies). It is a multigraph because there can be multiple edges of different types between any two nodes.

The point set representing the airplane in figure 13.2d can be converted into a graph by connecting each point to its K nearest neighbors. The result is a *geometric graph* where each point is associated with a position in 3D space. Figure 13.2e represents a *hierarchical graph*. The table, light, and room form three graphs, describing the adjacency of their respective components. These three graphs are themselves nodes in another graph that represents the topology of the objects in a larger model.

All types of graph can be processed using deep learning, but this chapter focuses on undirected graphs like the social network in figure 13.2a.

13.2 Graph representation

In addition to the graph structure itself, there is typically also information associated with each node. For example, in a social network, each individual might be characterized by a fixed length vector representing their interests. Sometimes, the edges also have information attached. For example, in the road network example each edge might be characterized by its length, number of lanes, frequency of accidents, and speed limit. The information at a node is stored in a *node embedding*, and the information at an edge is stored in an *edge embedding*.

More formally, a graph consists of a set of N nodes that are connected by a set of E

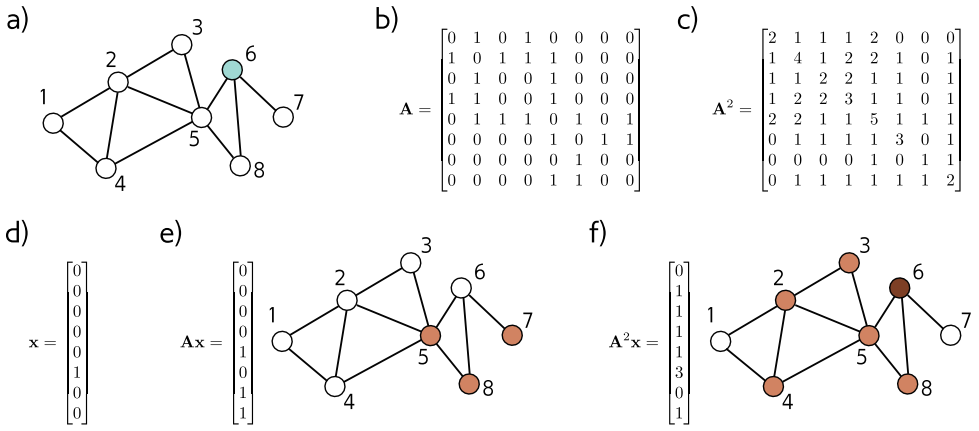

Figure 13.4 Properties of the adjacency matrix. a) Example graph. b) Position (m, n) of the adjacency matrix $\mathbf{A}$ contains the number of walks of length one from node m to node n. c) Position (m, n) of the squared adjacency matrix contains the number of walks of length two from node m to node n. d) One hot vector representing node six, which is highlighted in panel (a). e) When we pre-multiply this vector by $\mathbf{A}$, the result contains the number of walks of length one from node six to each node; we can reach nodes five, seven, and eight in one move. f) When we pre-multiply this vector by $\mathbf{A}^2$, the resulting vector contains the number of walks of length two from node six to each node; we can reach nodes two, three, four, five, and eight in two moves, and we can return to the original node in three different ways (via nodes five, seven, and eight).

edges. The graph is represented by three matrices $\mathbf{A}$, $\mathbf{X}$, and $\mathbf{E}$ representing the graph structure, node embeddings, and edge embeddings, respectively (figure 13.3).

The graph structure is represented by the *adjacency matrix*, $\mathbf{A}$. This is an $N \times N$ matrix where entry (m, n) is set to one if there is an edge between nodes m and n and zero otherwise. For undirected graphs, this matrix is always symmetric. For large sparse graphs, it can be stored as a list of connections (m, n) to save memory.

Problems 13.1–13.2

The n^{th} node has an associated node embedding $\mathbf{x}^{(n)}$ of length D . These embeddings are concatenated and stored in the $D \times N$ node data matrix $\mathbf{X}$. Similarly, the e^{th} edge has an associated edge embedding $\mathbf{e}^{(\mathbf{e})}$ of length D_E. These edge embeddings are collected into the $D_E \times E$ matrix $\mathbf{E}$. For simplicity, we'll initially consider graphs that only have node embeddings and return to edge embeddings in section 13.10.

13.2.1 Properties of the adjacency matrix

The adjacency matrix can be used to find the neighbors of a node using linear algebra. Consider encoding the n^{th} node as a one-hot column vector (a vector with only one non-zero entry at position n, which is set to one). When we pre-multiply this vector by

the adjacency matrix, it extracts the n^{th} column of the adjacency matrix and returns a vector with ones at the positions of the neighbors (i.e., all the places we can reach in a walk of length one from the n^{th} node). If we repeat this procedure (i.e., pre-multiply by $\mathbf{A}$ again), then the resulting vector contains the number of walks of length two from node n to every node (figures 13.4d–f).

Problems 13.3–13.4

In general, if we raise the adjacency matrix to the power of L, the entry at position (m, n) of $\mathbf{A}^L$ contains the number of unique "walks" of length L from node m to node n (figures 13.4a–c) This is not quite the same as the number of unique paths, since it includes routes that visit the same node more than once. Nonetheless, $\mathbf{A}^L$ still contains useful information about the graph connectivity; a non-zero entry at position (m, n) indicates that the distance from m to n must be less than or equal to that value.

13.2.2 Permutation of node indices

Node indexing in graphs is arbitrary; permuting the node indices results in a permutation of the columns of the node data matrix $\mathbf{X}$ and a permutation of both the rows and columns of the adjacency matrix $\mathbf{A}$. However, the underlying graph is unchanged (figure 13.5). This is in contrast to images, where permuting the pixels creates a different image, and to text, where permuting the words creates a different sentence.

The operation of exchanging node indices can be expressed mathematically by a *permutation matrix*, $\mathbf{P}$. This is a matrix where exactly one entry in each row and column take the value one and the remaining values are zero. When position (m, n) of the permutation matrix is set to one, it indicates that node m will become node n after the permutation. To map from one indexing to another, we use the operations:

Problem 13.5

$$\begin{aligned} \mathbf{X}' &= \mathbf{X}\mathbf{P} \\ \mathbf{A}' &= \mathbf{P}^T\mathbf{A}\mathbf{P}. \end{aligned} \tag{13.1}$$

It follows that any processing applied to the graph should also be indifferent to these permutations. Otherwise, the result will depend on the particular choice of node indices.

13.3 Graph neural networks, tasks, and loss functions

A graph neural network is a model that takes the node embeddings $\mathbf{X}$ and the adjacency matrix $\mathbf{A}$ as inputs and passes the graph through a series of K layers. At each layer the node embeddings are updated to create intermediate "hidden" representations $\mathbf{H}_k$, before finally computing output embeddings $\mathbf{H}_K$.

At the start of this network, each column of the input node embeddings $\mathbf{X}$ just contains information about the node itself. At the end, each column of the model output $\mathbf{H}_K$ contains information about the node and its context within the graph. This is similar to word embeddings passing through a transformer network, which represent words at the start, but represent the word meanings in the context of the sentence at the end.

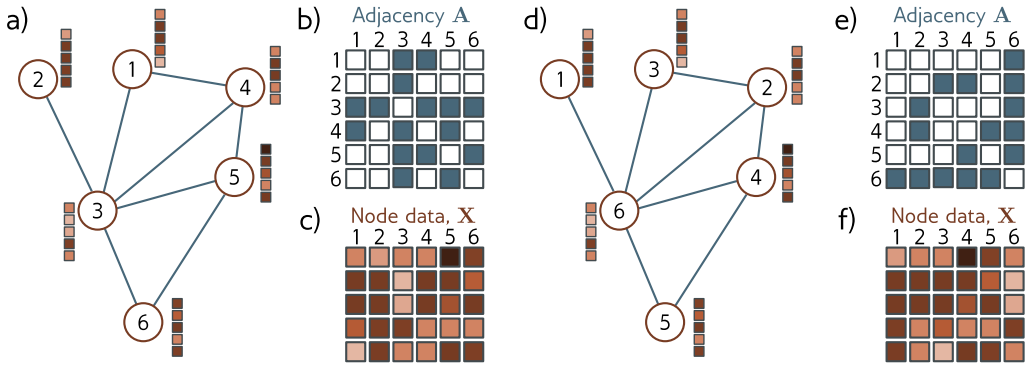

Figure 13.5 Permutation of node indices. a) Example graph, b) associated adjacency matrix, and c) node embeddings. d) The same graph where the (arbitrary) order of the indices has been changed. e) The adjacency matrix and f) node matrix are now different. This poses a challenge since any network layer that operates on the graph should be indifferent to the ordering of the nodes.

13.4 Tasks and loss functions

We defer discussion of the model until section 13.5, and first describe the types of problem graph neural networks tackle and the loss functions associated with each. Typically, supervised graph problems fall into one of three categories (figure 13.6).

Graph-level tasks: The network assigns a label or estimates one or more values from the entire graph, exploiting both the structure and node embeddings. For example, we might want to predict the temperature at which a molecule becomes liquid (a regression task) or whether a molecule is poisonous to human beings or not (a classification task).

For graph-level tasks, the output node embeddings are combined (e.g., by averaging them) and the resulting vector is mapped via a linear transformation or neural network to a vector of fixed-size. For regression, the mismatch between the result and the ground truth values is computed using the least squares loss. For classification, the output is passed through a sigmoid or softmax function and the binary or multiclass cross entropy loss is applied. For example, in binary classification, the output $y \in [0, 1]$ might be:

$$y = \text{sig}\left[\beta_K + \boldsymbol{\omega}_K \mathbf{H}_K \mathbf{1}/N\right], \tag{13.2}$$

where the scalar $\boldsymbol{\beta}_K$ and $1 \times D$ vector $\boldsymbol{\omega}_K$ are learned parameters. Post-multiplying the output embedding matrix $\mathbf{H}_K$ by the column vector $\mathbf{1}$ that contains ones has the effect of summing together all the embeddings and subsequently dividing by N computes the average. This is known as *mean pooling*.

Node-level tasks: The network assigns a label (classification) or one or more values (regression) to each node of the graph, using both the graph structure and node em-

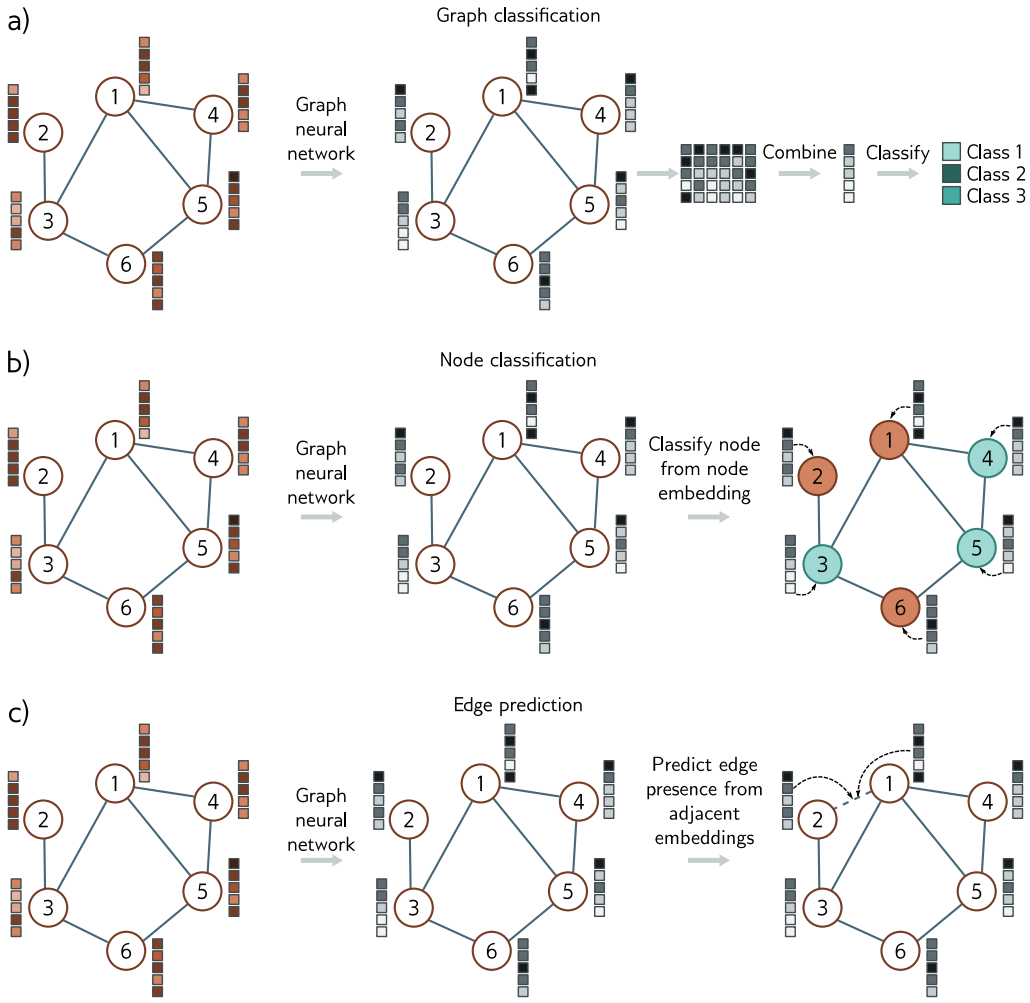

Figure 13.6 Common tasks for graphs. In each case, the input is a graph as represented by the adjacency matrix and node embeddings. The graph neural network processes the node embeddings by passing them through a series of layers. The node embeddings at the last layer contain information about both the node and its context in the graph. a) Graph classification. The node embeddings are combined (e.g., by averaging) and then mapped to a fixed-size vector that is passed through a softmax function to produce class probabilities. b) Node classification. Each node embedding is used individually as the basis for classification (cyan and orange colors represent assigned node classes). c) Edge prediction. Node embeddings adjacent to the edge are combined (e.g., by taking the dot product) to compute a single number that is mapped via a sigmoid function to produce a probability that a missing edge should be present.

beddings. For example, given a graph constructed from a 3D point cloud similar to figure 13.2d, the goal might be to classify the nodes according to whether they belong to the wing, fuselage, or engines. Loss functions are defined in exactly the same way as for graph-level tasks, except that now this is done independently at each node:

$$y^{(n)} = \text{sig}\left[\beta_K + \boldsymbol{\omega}_K \mathbf{h}_K^{(n)}\right]. \tag{13.3}$$

Edge prediction tasks: The network predicts whether there should be an edge or not between nodes n and m. For example, in the social network setting, the network might predict whether two people know and like each other and suggest that they connect if that is the case. This is a binary classification task where the two node embeddings must be mapped to a single number representing the probability that the edge is present. One possibility is to take the dot product of the node embeddings and pass the result through a sigmoid function to create the probability:

$$y^{(mn)} = \text{sig}\left[\mathbf{h}^{(m)T}\mathbf{h}^{(n)}\right]. \tag{13.4}$$

13.5 Graph convolutional networks

There are many types of graph neural networks but here we focus on *spatial-based convolutional graph neural networks* or *GCNs* for short. These models are convolutional in that they update each node by aggregating information from nearby nodes, and as such, they induce a *relational inductive bias*. They are spatial-based because they do this in a straightforward manner using the original graph structure. This is in contrast to spectral-based methods that apply convolutions in the Fourier domain.

Each layer of the GCN is a function $\mathbf{F}[\bullet]$ with parameters $\boldsymbol{\Phi}$ that takes the node embeddings and adjacency matrix and outputs new node embeddings. The network can hence be written as:

$$\begin{aligned}
\mathbf{H}_1 &= \mathbf{F}[\mathbf{X}, \mathbf{A}, \boldsymbol{\phi}_0] \\
\mathbf{H}_2 &= \mathbf{F}[\mathbf{H}_1, \mathbf{A}, \boldsymbol{\phi}_1] \\
\mathbf{H}_3 &= \mathbf{F}[\mathbf{H}_2, \mathbf{A}, \boldsymbol{\phi}_2] \\
\vdots &= \vdots \\
\mathbf{H}_K &= \mathbf{F}[\mathbf{H}_{K-1}, \mathbf{A}, \boldsymbol{\phi}_{K-1}],
\end{aligned} \tag{13.5}$$

where $\mathbf{X}$ is the input, $\mathbf{A}$ is the adjacency matrix, $\mathbf{H}_k$ contains the modified node embeddings at the k^{th} layer, and $\boldsymbol{\phi}_k$ denotes the parameters associated with that layer.

13.5.1 Equivariance and invariance

We noted before that the indexing of the nodes in the graph is arbitrary and any permutation of the node indices does not change the graph. It is hence imperative that any model respects this property. It follows that each layer must be equivariant (see section 10.1) with respect to permutations of the node indices. In other words, if we permute the node indices, the node embeddings at each stage will be permuted in the same way. In mathematical terms, if $\mathbf{P}$ is a permutation matrix then we must have:

$$\mathbf{H}_{k+1}\mathbf{P} = \mathbf{F}[\mathbf{H}_k\mathbf{P}, \mathbf{P}^T\mathbf{A}\mathbf{P}, \boldsymbol{\phi}_k]. \tag{13.6}$$

For node classification and edge prediction tasks, the output should also be equivariant with respect to permutations of the node indices. However, for graph-level tasks, the final layer aggregates information from across the graph so the output is invariant to the order of the nodes. In fact, the output layer from equation 13.2 achieves this because:

Problem 13.6

$$y = \text{sig}\left[\beta_K + \boldsymbol{\omega}_K\mathbf{H}_K\mathbf{1}/N\right] = \text{sig}\left[\beta_K + \boldsymbol{\omega}_K\mathbf{H}_K\mathbf{P}\mathbf{1}/N\right], \tag{13.7}$$

for any permutation matrix $\mathbf{P}$.

This mirrors the case for images, where segmentation should be equivariant to image transformations and image classification should be invariant (figure 10.1). There is no known way to guarantee these properties exactly for images, although convolutional and pooling layers partially achieve this with respect to translations. However, for graphs, it is possible to define networks that ensure equivariance or invariance to permutations.

13.5.2 Parameter sharing

Chapter 10 argued that it isn't sensible to apply fully connected networks to images because this requires the network to learn how to recognize an object separately at every image position. Instead, we used convolutional layers that processed every position in the image identically. This reduced the number of parameters and introduced an inductive bias that forced the model to treat every part of the image in the same way.

The same argument can be made about nodes in a graph. We could learn a model with separate parameters associated with each node. However, now the network must relearn the meaning of the connections in the graph at each position, and training would require many graphs with the same topology. Instead, we build a model that uses the same parameters at every node, reducing the number of parameters dramatically, and sharing what the network has learned at each node across the entire graph.

Recall that a convolution (equation 10.3) updates a variable by taking a weighted sum of information from its neighbors. One way to think of this is that each neighbor sends a message to the variable of interest, which aggregates these messages to form the update. When we considered images, the neighbors were pixels from a fixed-size square region around the current position, and so the spatial relationships at each position are the same. However, in a graph, each node may have a different number of neighbors and there are no consistent relationships; there is no sense that we can weight information from a node that is "above" this one differently from a node that is "below" it.

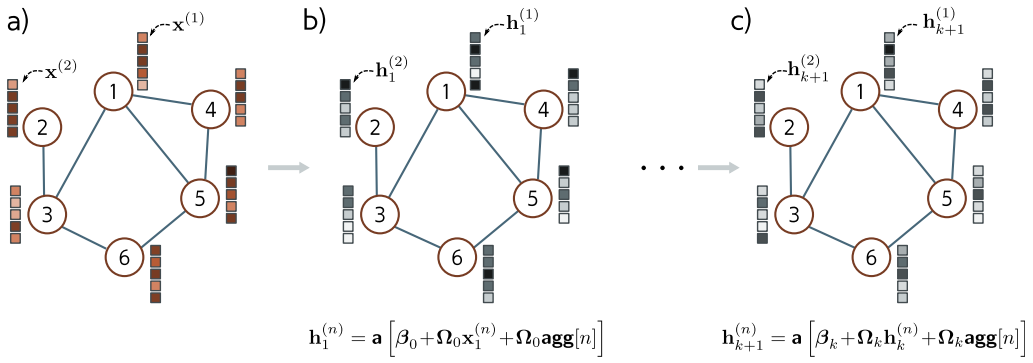

Figure 13.7 Simple Graph CNN layer. a) Input graph consists of structure (embodied in graph adjacency matrix $\mathbf{A}$, not shown) and node embeddings (stored in columns of $\mathbf{X}$). b) Each node in the first hidden layer is updated by (i) aggregating the neighboring nodes to form a single vector, (ii) applying a linear transformation $\boldsymbol{\Omega}_0$ to the aggregated nodes, (iii) applying the same linear transformation $\boldsymbol{\Omega}_0$ to the original node, (iv) adding these together with a bias $\boldsymbol{\beta}_0$, and finally (v) applying a nonlinear activation function $\mathbf{a}[\bullet]$ like a ReLU. This process is repeated at subsequent layers (but with different parameters for each layer) until we produce the final embeddings at the end of the network.

13.5.3 Example GCN layer

These considerations lead to a simple GCN layer (figure 13.7). At each node n in layer k we aggregate information from neighboring nodes by summing their node embeddings $\mathbf{h}_\bullet$:

$$\mathbf{agg}[n,k] = \sum_{m \in \text{ne}[n]} \mathbf{h}_k^{(m)}, \tag{13.8}$$

where $\text{ne}[n]$ returns the set of indices of the neighbors of node n. Then we apply a linear transformation to the embedding $\mathbf{h}_k^{(n)}$ at the current node and to this aggregated value, add a bias term, and pass the result through a nonlinear activation function $\mathbf{a}[\bullet]$, which is applied independently to every member of its vector argument:

$$\mathbf{h}_{k+1}^{(n)} = \mathbf{a}\left[\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \cdot \mathbf{h}_k^{(n)} + \boldsymbol{\Omega}_k \cdot \mathbf{agg}[n,k]\right]. \tag{13.9}$$

We can write this more succinctly by noting that post-multiplication of a matrix by a vector returns a weighted sum of its columns. The n^{th} column of the adjacency matrix $\mathbf{A}$ contains ones at the positions of the neighbors. Hence, if we collect the node embeddings into the $D \times N$ matrix $\mathbf{H}_k$ and post-multiply by the adjacency matrix $\mathbf{A}$, the n^{th} column of the result is $\mathbf{agg}[n,k]$. The update for the nodes is now:

$$\begin{aligned}\mathbf{H}_{k+1} &= \mathbf{a}\left[\boldsymbol{\beta}_k\mathbf{1}^T + \boldsymbol{\Omega}_k\mathbf{H}_k + \boldsymbol{\Omega}_k\mathbf{H}_k\mathbf{A}\right] \\ &= \mathbf{a}\left[\boldsymbol{\beta}_k\mathbf{1}^T + \boldsymbol{\Omega}_k\mathbf{H}_k(\mathbf{A}+\mathbf{I})\right], \end{aligned} \tag{13.10}$$

where $\mathbf{1}$ is an N×1 vector containing ones. Here, the nonlinear activation function $\mathbf{a}[\bullet]$ is applied independently to every member of its matrix argument.

Problem 13.7

This layer satisfies the design considerations: it is equivariant to permutations of the node indices, can cope with any number of neighbors, exploits the structure of the graph to provide a relational inductive bias, and shares parameters throughout the graph.

13.6 Example: graph classification

We now combine these ideas to describe a network that classifies molecules as being toxic or harmless. The input to the network is the adjacency matrix $\mathbf{A}$ and node embedding matrix $\mathbf{X}$. The adjacency matrix derives directly from the molecular structure. Each node embedding takes one of 118 possible values corresponding to the 118 elements of the periodic table. These are encoded as one-hot vectors (i.e., vectors of length 118 where every position is zero except for the position corresponding to the relevant chemical element, which is set to one). These one-hot vectors form the columns of the data matrix $\mathbf{X} \in \mathbb{R}^{118\times N}$. The node embeddings can be transformed to an arbitrary size D by the first weight matrix $\boldsymbol{\Omega}_0 \in \mathbb{R}^{D\times 118}$.

The network equations are:

$$\begin{aligned}\mathbf{H}_1 &= \mathbf{a}\left[\boldsymbol{\beta}_0\mathbf{1}^T + \boldsymbol{\Omega}_0\mathbf{X}(\mathbf{A}+\mathbf{I})\right] \\ \mathbf{H}_2 &= \mathbf{a}\left[\boldsymbol{\beta}_1\mathbf{1}^T + \boldsymbol{\Omega}_1\mathbf{H}_1(\mathbf{A}+\mathbf{I})\right] \\ \vdots &= \vdots \\ \mathbf{H}_K &= \mathbf{a}\left[\boldsymbol{\beta}_{K-1}\mathbf{1}^T + \boldsymbol{\Omega}_{K-1}\mathbf{X}_{k-1}(\mathbf{A}+\mathbf{I})\right] \\ \mathrm{f}[\mathbf{X},\mathbf{A},\boldsymbol{\Phi}] &= \mathrm{sig}\left[\beta_K + \boldsymbol{\omega}_K\mathbf{H}_K\mathbf{1}/N\right], \end{aligned} \tag{13.11}$$

where the output $\mathrm{f}[\mathbf{X},\mathbf{A},\boldsymbol{\Phi}]$ of the network is a single value that determines the probability that the molecule is toxic (see equation 13.2).

13.6.1 Training with batches

Given a dataset of I training graphs $\{\mathbf{X}_i,\mathbf{A}_i\}$ and their labels y_i, the parameters $\boldsymbol{\Phi} = \{\boldsymbol{\beta}_K,\boldsymbol{\Omega}_K\}_{k=0}^K$ can be trained using SGD and the binary cross entropy loss (equation 5.22). Previously, we have exploited the parallelism of modern hardware to concurrently process the entire batch simultaneously. However, it's not clear how to do this here, since each graph may have a different number of nodes. Hence, the matrices $\mathbf{X}_i$ and $\mathbf{A}_i$ have different sizes and there is no way to concatenate them into 3D tensors.

Luckily, there is a simple trick that allows us to perform the forward and backward passes in parallel. The graphs in the batch are treated as disjoint components of a single large graph. The network can then be run as a single instance of the network equations. The mean pooling is carried out only over the individual graphs to make a single representation per graph that can be fed into the loss function.

13.7 Inductive vs. transductive models

All of the models described so far in this book have been *inductive*: we exploit a training set of labeled data to learn the relation between the inputs and outputs. Then we apply this to new test data. One way to think of this is that we are learning the rule that maps inputs to outputs and then applying it elsewhere.

By contrast, a *transductive* model considers both the labeled and unlabeled data at the same time. It does not produce a rule, but merely a labeling for all of the unknown outputs. This is sometimes termed *semi-supervised learning*. It has the advantage that it can use patterns in the unlabeled data to help make its decisions. However, it has the disadvantage that when extra unlabeled data are added, the model should be retrained.

Both inductive and transductive problems exist for graphs (figure 13.8). Sometimes we have many labeled graphs from which we can learn a relation between the graph and the labels. For example, we might have many molecules, each of which is labeled according to whether it is toxic to humans. We learn the rule that maps the graph to the toxic/non-toxic label and then apply this to new molecules. However, in other cases like the scientific citations example, there is a single monolithic graph. We might have labels indicating the field (physics, biology, etc.) for some of the nodes, and wish to label the remaining nodes. Here, the training and test data are irrevocably connected.

Graph-level tasks only occur in the inductive setting where there are training and test graphs. However, node-level tasks and edge prediction tasks can occur in either setting. In the transductive case, the loss function minimizes the mismatch between the model output and the ground truth where this is known. New predictions are computed by running the forward pass and retrieving the results where the ground truth is unknown.

13.8 Transductive learning example: node classification

As a second example, consider a binary node classification task in a transductive setting. We start with a commercial-sized graph with millions of nodes. Some of these nodes have ground truth binary labels and the goal is to label the remaining unlabeled nodes. The body of the network will be the same as in the previous example (equation 13.11) but with a different final layer that produces an output vector of size $1 \times N$:

$$\mathbf{f}[\mathbf{X}, \mathbf{A}, \boldsymbol{\Phi}] = \mathbf{sig}\left[\beta_K \mathbf{1}^{\mathbf{T}} + \boldsymbol{\omega}_{\mathbf{K}} \mathbf{H}_{\mathbf{K}}\right], \tag{13.12}$$

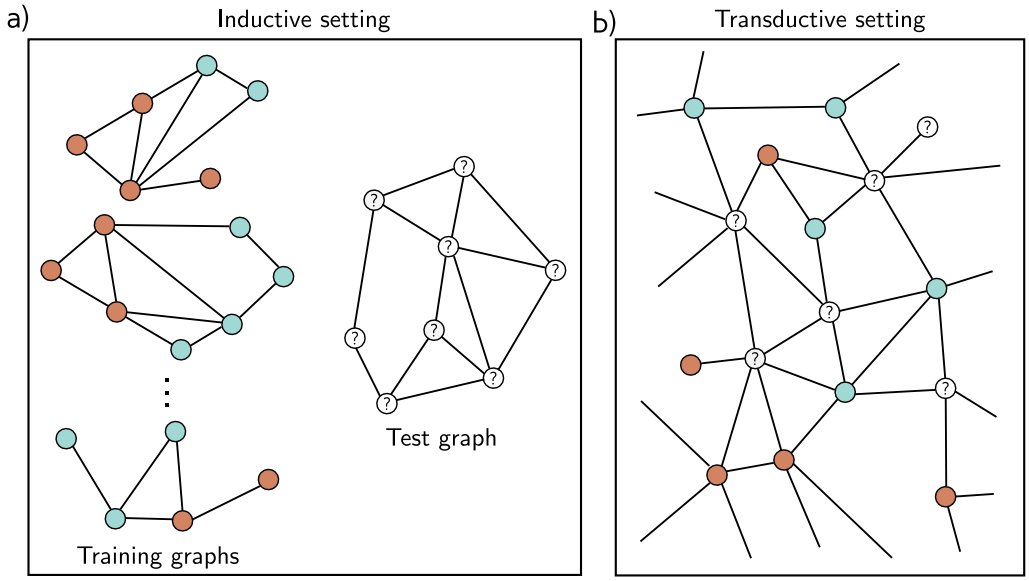

Figure 13.8 Inductive vs. transductive problems. a) Node classification task in inductive setting. We are given a set of I training graphs, where the node labels (orange and cyan colors) are known. After training, we are given a test graph and must assign labels to each node. b) Node classification in transductive setting. There is one large graph in which some of the nodes have labels (orange and cyan colors) and others are unknown. We train the model to predict the known labels correctly and then examine the predictions at the unknown nodes.

where the function $\mathbf{sig}[\bullet]$ applies the sigmoid function independently to every element of the row vector input. As usual, we use the binary cross-entropy loss, but now only at nodes where we know the ground truth label y. Note that equation 13.12 is just a vectorized version of the node classification loss from equation 13.3.

Training this network raises two problems. First, it is logistically difficult to train a graph neural network of this size. Consider that in the forward pass, we will have to store the node embeddings at every layer of the network. This will involve both storing and processing a structure that is several times the size of the entire graph, which may not be practical. Second, we have only a single graph, so it's not obvious how to perform stochastic gradient descent. How can we form a batch if there is only a single object?

13.8.1 Choosing batches

One way to form a batch is just to choose a random subset of the labeled nodes at each step of training. Each of these nodes depends on its neighbors in the previous layer.

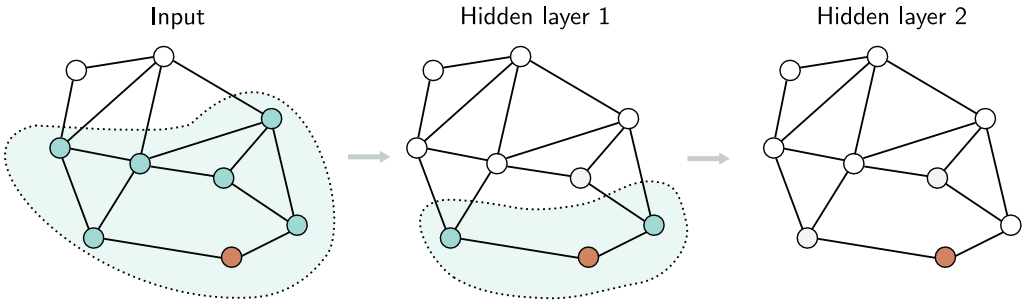

Figure 13.9 Receptive fields in graph neural networks. Consider the orange node in hidden layer two (right). This receives input from the nodes in the 1-hop neighborhood in hidden layer one (shaded region in center). The nodes in hidden layer one receive inputs from their neighbors in turn. This means that the orange node in layer two receives inputs from all the input nodes in the 2-hop neighborhood (shaded region on left). The region of the graph that contributes to a given node is equivalent to the notion of a receptive field in convolutional neural networks

These in turn depend on their neighbors in the layer before that, so we can think of each node as having the equivalent of a receptive field (figure 13.9). The size of the receptive field is referred to as the *k-hop neighborhood*. We can hence perform a gradient descent step using the graph that forms the union of the k-hop neighborhoods of the nodes in the batch; the remaining inputs do not contribute.

Unfortunately, this only works if there are relatively few layers and the graph is sparse. For densely connected networks with many layers, every node may be in the receptive field of every output, and so this may not reduce the graph size at all. This is known as the *graph expansion problem*. Two approaches that tackle this problem are *neighborhood sampling* and *graph partitioning*

Neighborhood sampling: The full graph that feeds into the batch of nodes is sampled, thereby reducing the possible number of connections at each network layer (figure 13.10). For example, we might start with the batch nodes and randomly sample a fixed number of their neighbors in the previous layer. Then, we randomly sample a fixed number of *their* neighbors in the layer before, and so on. The graph still increases in size with each layer, but in a way that is much more controlled. This is done anew for each batch, so even if the same batch is drawn twice, the contributing neighbors differ. This is also reminiscent of dropout (section 9.3.3) and adds some regularization.

Graph partitioning: A second approach is to cluster the original graph into disjoint subsets of nodes (i.e., smaller graphs that are not connected to one another) before processing (figure 13.11). There are standard algorithms to choose these subsets so that the number of internal links is maximized. These smaller graphs can each be treated as

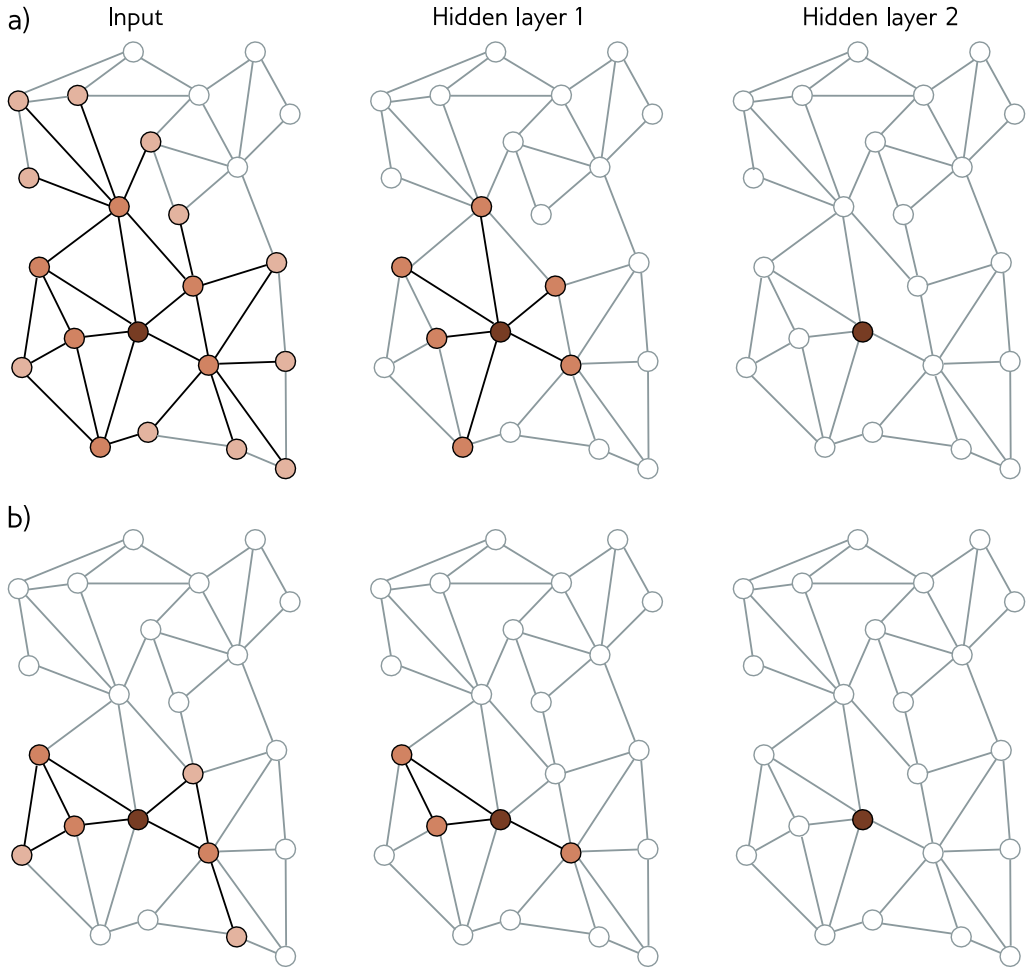

Figure 13.10 Neighborhood sampling. a) One way of forming batches on large graphs is to choose a subset of labeled nodes in the output layer (here just one node in layer two, right) and then work back to find all of the nodes in the K-hop neighborhood (receptive field). Only this sub-graph is needed to train this batch. Unfortunately, if the graph is densely connected, this may retain a large proportion of the graph. b) One solution is neighborhood sampling. As we work back from the final layer, we select a subset of neighbors (here three) in the layer before, and a subset of the neighbors of these in the layer before that. This restricts the size of the graph for training the batch.

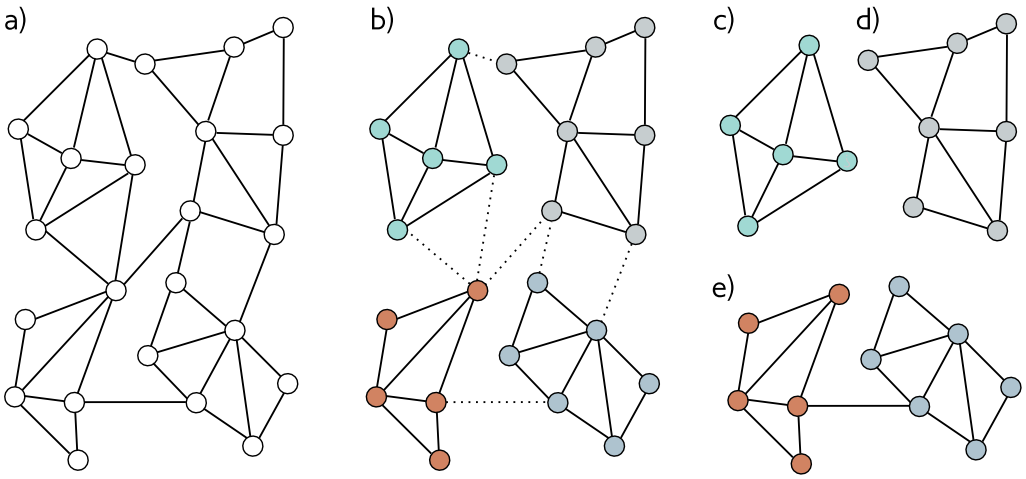

Figure 13.11 Graph partitioning. a) Input graph. b) The input graph is partitioned into smaller subgraphs, using a principled method that removes the fewest edges. c-d) We can now use these subgraphs as batches to train in a transductive setting, so here there are four possible batches. e) Alternatively, we can use combinations of the subgraphs as batches, reinstating the edges between them. If we use pairs of subgraphs, there would be six possible batches here.

batches, or a random combination of them can be combined to form a batch (reinstating any links between them from the original graph).

Given one of the above methods to form batches, we can now train the network parameters in the same way as for the inductive setting, dividing the labeled nodes into train, test, and validation sets as desired; we have effectively converted a transductive problem to an inductive one. To perform inference, we compute predictions for the unknown nodes based on their k-hop neighborhood. Unlike training, this does not require storing the intermediate representations, and so is much more memory efficient.

13.9 Layers for graph convolutional networks

In the previous examples, we combined messages from adjacent nodes by summing them together with the transformed current node. This was accomplished by post-multiplying the node embedding matrix $\mathbf{H}$ by the adjacency matrix plus the identity $\mathbf{A}+\mathbf{I}$. We now consider different approaches to both (i) the combination of the current embedding with the aggregated neighbors and (ii) the aggregation process itself.

13.9.1 Combining current node and aggregated neighbors

In the example GCN layer above, we combined the aggregated neighbors $\mathbf{HA}$ with the current nodes $\mathbf{H}$ by just summing them:

$$\mathbf{H}_{k+1} = \mathbf{a}\left[\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \mathbf{H}_k(\mathbf{A}+\mathbf{I})\right]. \tag{13.13}$$

In another variation, the current node is multiplied by a factor of $(1+\epsilon_k)$ before contributing to the sum, where ϵ_k is a learned scalar that is different for each layer:

$$\mathbf{H}_{k+1} = \mathbf{a}\left[\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \mathbf{H}_k(\mathbf{A}+(1+\epsilon_k)\mathbf{I})\right]. \tag{13.14}$$

This is known as *diagonal enhancement*. A further possibility is to treat the current node differently from the aggregated neighbors by applying a different linear transform $\boldsymbol{\Psi}_k$ to the current node:

$$\begin{aligned}\mathbf{H}_{k+1} &= \mathbf{a}\left[\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Psi}_k \mathbf{H}_k + \boldsymbol{\Omega}_k \mathbf{H}_k \mathbf{A}\right] \\ &= \mathbf{a}\left[\boldsymbol{\beta}_k \mathbf{1}^T + \begin{bmatrix}\boldsymbol{\Omega}_k & \boldsymbol{\Psi}_k\end{bmatrix}\begin{bmatrix}\mathbf{H}_k\mathbf{A} \\ \mathbf{H}_k\end{bmatrix}\right] \\ &= \mathbf{a}\left[\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}'_k \begin{bmatrix}\mathbf{H}_k\mathbf{A} \\ \mathbf{H}_k\end{bmatrix}\right],\end{aligned} \tag{13.15}$$

where we have defined $\boldsymbol{\Omega}'_k = \begin{bmatrix}\boldsymbol{\Omega}_k & \boldsymbol{\Psi}_k\end{bmatrix}$ in the third line.

Problem 13.9

13.9.2 Residual connections

With residual connections, the aggregated representation from the neighbors is transformed and passed through the activation function before summation or concatenation with the current node. For the latter case, the associated network equations are:

$$\mathbf{H}_{k+1} = \begin{bmatrix}\mathbf{a}\left[\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \mathbf{H}_k \mathbf{A}\right] \\ \mathbf{H}_k\end{bmatrix}. \tag{13.16}$$

All of the above methods aggregate the neighbors by summing the node embeddings. In the next three sections, we introduce three different approaches to aggregation.

13.9.3 Mean aggregation

Sometimes it's better to take the average of the neighbors rather than the sum; this might be superior if the embedding information is more important and the structural information is less important since the magnitude of the neighborhood contributions will not depend on the number of neighbors:

$$\mathbf{agg}[n] = \frac{1}{|\mathrm{ne}[n]|}\sum_{m\in \mathrm{ne}[n]} \mathbf{h}_m, \tag{13.17}$$

where as before ne[n] denotes a set containing the indices of the neighbors of the n^{th} node. Equation 13.17 can be computed neatly in matrix form by introducing the diagonal $N \times N$ degree matrix $\mathbf{D}$. Each non-zero element of this matrix contains the number of neighbors for the associated node. It follows that each diagonal element in the inverse matrix $\mathbf{D}^{-1}$ contains the denominator that we need to compute the average. The new GCN layer can be written as:

Problem 13.8

$$\mathbf{H}_{k+1} = \mathbf{a}\left[\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \mathbf{H}_k(\mathbf{A}\mathbf{D}^{-1} + \mathbf{I})\right]. \tag{13.18}$$

13.9.4 Kipf normalization

There are many variations of graph neural networks based on mean aggregation. Sometimes the current node is included with its neighbors in the mean computation, rather than treated separately. In Kipf normalization, the sum of the node representations is normalized as:

Problem 13.9

$$\mathbf{agg}[n] = \sum_{m \in \text{ne}[n]} \frac{\mathbf{h}_m}{\sqrt{|\text{ne}[n]||\text{ne}[m]|}}, \tag{13.19}$$

with the logic that information coming from nodes with a very large number of neighbors should be downweighted since there are many connections and they provide less unique information. This can also be expressed in matrix form using the degree matrix:

$$\mathbf{H}_{k+1} = \mathbf{a}\left[\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \mathbf{H}_k(\mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2} + \mathbf{I})\right]. \tag{13.20}$$

13.9.5 Max pooling aggregation

An alternative operation that is also invariant to permutation is computing the maximum of a set of objects. The *max pooling* aggregation operator is:

$$\mathbf{agg}[n] = \underset{m \in \text{ne}[n]}{\mathbf{max}}\left[\mathbf{h}_m\right], \tag{13.21}$$

where the operator $\mathbf{max}[\bullet]$ returns the element-wise maximum of the vectors $\mathbf{h}_m$ that are neighbors to the current node n.

13.9.6 Aggregation by attention

The aggregation methods discussed so far either weight the contribution of the neighbors equally or in a way that depends on the graph topology. Conversely, in *graph attention layers*, the weights depend on the data at the nodes. A linear transform is applied to the current node embeddings so that:

$$\mathbf{H}'_k = \boldsymbol{\beta}_k \mathbf{1}^{\mathbf{T}} + \boldsymbol{\Omega}_{\mathbf{k}} \mathbf{H}. \tag{13.22}$$

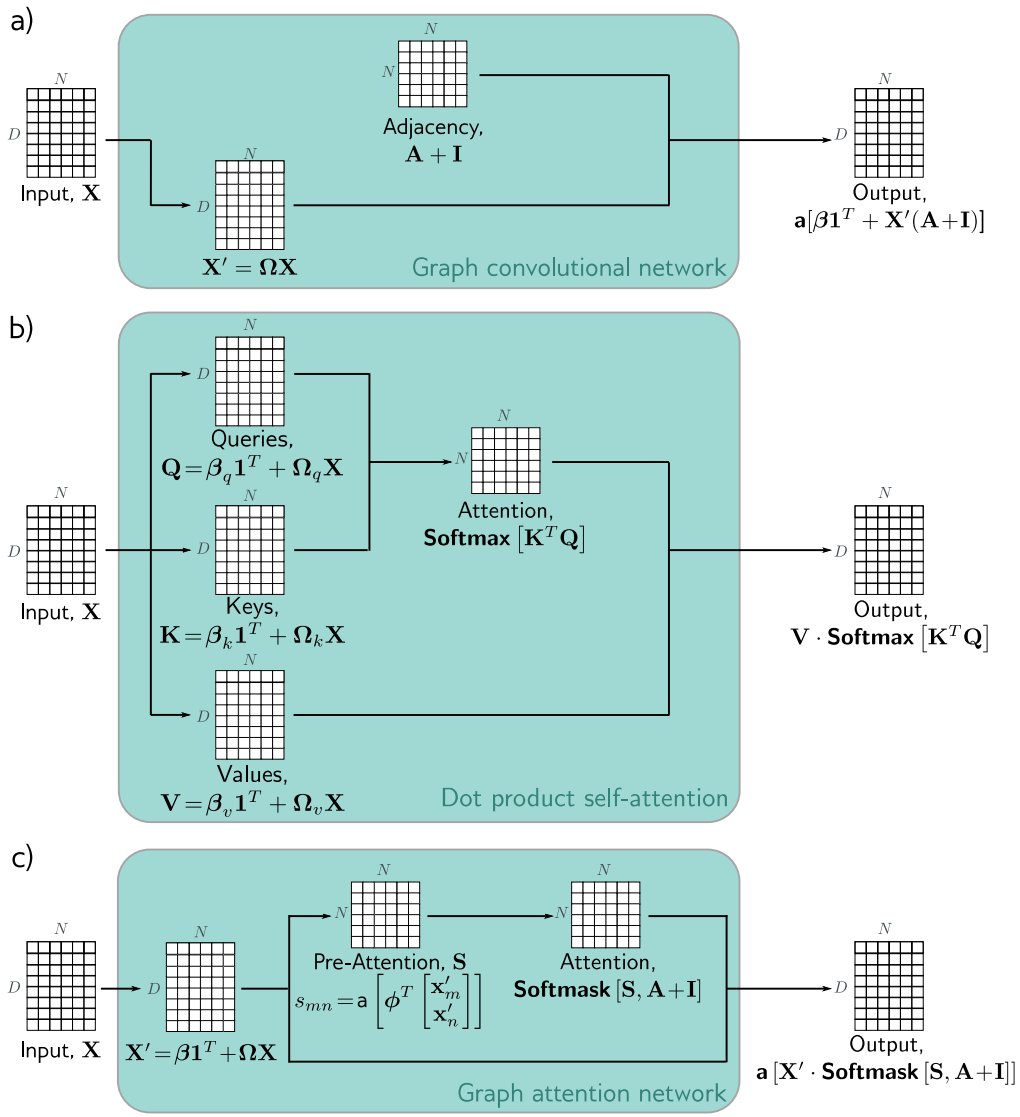

Figure 13.12 Comparison of graph convolutional network, dot product attention, and graph attention network. In each case, the mechanism maps N embeddings of size D stored in a $D \times N$ matrix $\mathbf{X}$ to an output of the same size. The graph convolutional network applies a linear transformation $\mathbf{X}' = \mathbf{\Omega X}$ to the data matrix. It then computes a weighted sum of the transformed data, where the weighting is based on the adjacency matrix. A bias $\boldsymbol{\beta}$ is added and the result is passed through an activation function. b) The outputs of the self-attention mechanism are also weighted sums of the transformed inputs, but this time the weights depend on the data itself via the attention matrix. c) The graph attention network combines both of these mechanisms; the weights are both computed from the data and based on the adjacency matrix.

Then the similarity s_{mn} of each transformed node embedding $\mathbf{h}'_m$ to the transformed node embedding $\mathbf{h}'_m$ is computed by concatenating the pairs, taking a dot product with the column vector $\boldsymbol{\phi}_k$, and applying an activation function:

$$s_{mn} = \mathrm{a}\left[\boldsymbol{\phi}_k^T \begin{bmatrix}\mathbf{h}'_m \\ \mathbf{h}'_n\end{bmatrix}\right]. \tag{13.23}$$

These variables are stored in an $N \times N$ matrix $\mathbf{S}$, where each element represents the similarity of every node to every other. However, only those values corresponding to the current node and its neighbors are required. This can be achieved by pointwise multiplying $\mathbf{S}$ with $\mathbf{A}+\mathbf{I}$. As in dot-product self-attention, the attention weights that contribute to each output embedding are normalized to be positive and sum to one using the softmax operation. These weights are applied to the transformed embeddings:

$$\mathbf{H}_{k+1} = \mathbf{a}\left[\mathbf{H}'_k \cdot \mathbf{Softmask}[\mathbf{S}, \mathbf{A}+\mathbf{I}]\right], \tag{13.24}$$

where $\mathbf{a}[\bullet]$ is a second activation function. The function $\mathbf{Softmask}[\bullet,\bullet]$ applies the softmax operation separately to each column of its first argument, setting values where the second argument is zero to negative infinity. This has the effect of ensuring that the attention to non-neighboring nodes is zero.

This is very similar to the self-attention computation in transformers (figure 13.12), except that (i) the keys, queries, and values are all the same (ii) the measure of similarity is different, and (iii) the attentions are masked so that each node only attends to itself and its neighbors. As in transformers, this system can be extended to use multiple heads that are run in parallel and recombined.

Problem 13.10

13.10 Edge graphs

Until now, we have focused on processing node embeddings, which evolve as they are passed through the network so that by the end of the network they represent both the node and its context in the graph. We now consider the case where the information is associated with the edges of the graph.

It is easy to adapt the machinery for node embeddings to process edge embeddings using the *edge graph* (also known as the *adjoint graph* or *line graph*). This is a complementary graph, in which each edge in the original graph becomes a node, and every two edges that have a node in common in the original graph create an edge in the new graph (figure 13.13). In general, a graph can be recovered from its edge graph, and so it's possible to swap between these two representations.

Problems 13.11–13.13

To process edge embeddings, the graph is translated to its edge graph and we use exactly the same techniques, aggregating information at each new node from its neighbors and combining this with the current representation. When both node and edge embeddings are present, we can translate back and forth between the two graphs. Now there are four possible updates (nodes update nodes, nodes update edges, edges update nodes, and edges update edges) and these can be alternated as desired, or with minor modifications, nodes can be updated simultaneously from both nodes and edges.

Problem 13.14

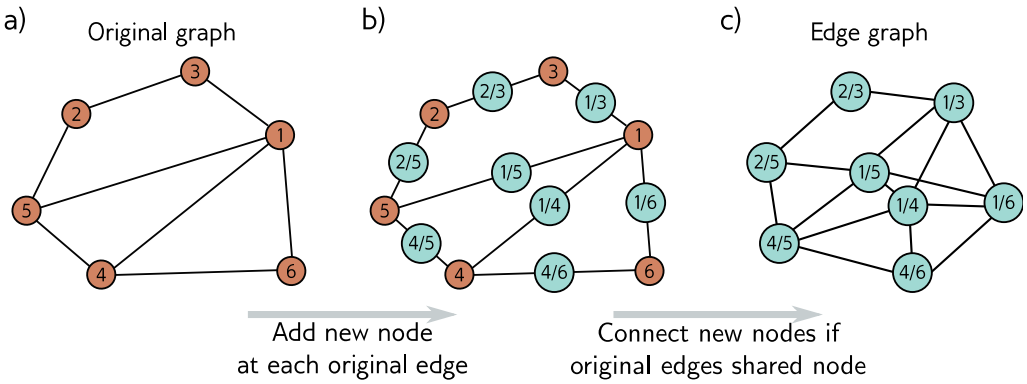

Figure 13.13 Edge graph. a) Graph with six nodes. b) To create the edge graph, we assign one node for each original edge (cyan circles), and c) connect the new nodes if the edges they represent connect to the same node in the original graph.

13.11 Summary

Graphs consist of a set of nodes, where pairs of these nodes are connected by edges. Both nodes and edges can have data attached and these are referred to as node embeddings and edge embeddings, respectively. Many real-world problems can be framed in terms of graphs, where the goal is to establish a property of the entire graph, properties of each node or edge, or the presence of additional edges in the graph.

Graph neural networks are deep learning models that are applied to graphs. Since the node order in graphs is arbitrary, the layers of graph neural networks must be equivariant to permutations of the node indices. Spatial-based convolutional networks are a family of graph neural networks that aggregate information from the neighbors of a node and then use this to update the node embeddings.

One challenge of processing graphs is that they often occur in the transductive setting, where there is only one partially labeled graph rather than sets of training and test graphs. This graph is often extremely large, which adds further challenges in terms of training, and has led to sampling and partitioning algorithms. The edge graph has a node for every edge in the original graph. By converting to this representation, graph neural networks can be used to update the edge embeddings.

Notes

Sanchez-Lengeling et al. (2021) and Daigavane et al. (2021) present good introductory articles on graph processing using neural networks. Recent surveys of research in graph neural networks can be found in articles by Zhou et al. (2020a) and Wu et al. (2020d) and the books of Hamilton

(2020) and Ma & Tang (2021). GraphEDM (Chami et al. 2020) unifies many existing graph algorithms into a single framework. In this chapter, we have related graphs to convolutional networks following Bruna et al. (2013), but there are also strong connections with belief propagation (Dai et al. 2016) and graph isomorphism tests (Hamilton et al. 2017a). Zhang et al. (2019b) provide a review that focuses specifically on graph convolutional networks. Bronstein et al. (2021) provide a general overview of geometric deep learning including learning on graphs.

Applications: Applications include graph classification (e.g., Zhang et al. 2018b), node classification (e.g., Kipf & Welling 2016a), edge prediction (e.g., Zhang & Chen 2018), graph clustering (e.g., Tsitsulin et al. 2020), and recommender systems (e.g., Wu et al. 2020c). Node classification methods are reviewed in Xiao et al. (2022), graph classification methods in Errica et al. (2019), and edge prediction methods in Mutlu et al. (2020) and Kumar et al. (2020).

Graph neural networks: Graph neural networks were introduced by Gori et al. (2005) and Scarselli et al. (2008) who formulated them as a generalization of recursive neural networks. The latter model used the iterative update:

$$\mathbf{h}_n \leftarrow \mathbf{f}\left[\mathbf{x}_n, \mathbf{x}_{m\in \text{ne}[n]}, \mathbf{e}_{e\in \text{nee}[n]}, \mathbf{h}_{m\in \text{ne}[n]}, \boldsymbol{\phi}\right], \tag{13.25}$$

in which each node embedding $\mathbf{h}_n$ is updated based on the initial embedding $\mathbf{x}_n$, initial embeddings $\mathbf{x}_{m\in \text{ne}[n]}$ at the neighboring nodes, initial embeddings $\mathbf{e}_{e\in \text{nee}[n]}$ at the neighboring edges, and neighboring node embeddings $\mathbf{h}_{m\in \text{ne}[n]}$. For convergence, the function $\mathbf{f}[\bullet, \bullet, \bullet, \bullet, \boldsymbol{\phi}]$ must be a contraction mapping (see figure **??**). If we unroll this equation in time and for K steps and allow different parameters $\boldsymbol{\phi}_k$ at each time K, then it becomes similar to the graph convolutional network formulation. Subsequent work extended graph neural networks to use gated recurrent units (Li et al. 2015) and long short-term memory networks (Selsam et al. 2018).

Spectral methods: Bruna et al. (2013) applied the convolution operation in the Fourier domain. The Fourier basis vectors can be found by taking the eigendecomposition of the *graph Laplacian matrix*, $\mathbf{L} = \mathbf{D} - \mathbf{A}$ where $\mathbf{D}$ is the degree matrix and $\mathbf{A}$ is the adjacency matrix. This has the disadvantages that the filters are not localized and the decomposition is prohibitively expensive for large graphs. Henaff et al. (2015) tackled the first problem by forcing the Fourier representation to be smooth (and hence the spatial domain to be localized). Defferrard et al. (2016) introduced ChebNet, which approximates the filters efficiently by making use of the recursive properties of Chebyshev polynomials. This both provides spatially localized filters and reduces the computation. Kipf & Welling (2016a) simplified this further to construct filters that use only a 1-hop neighborhood, resulting in a formulation that is similar to the spatial methods described in this chapter and providing a bridge between spectral and spatial methods.

Spatial methods: Spectral methods are ultimately based on the Graph Laplacian, and so if the graph changes, the model must be retrained. This problem spurred the development of spatial methods. Duvenaud et al. (2015) defined convolutions in the spatial domain, using a different weight matrix to combine the adjacent embeddings for each node degree. This has the disadvantage that it becomes impractical if some nodes have a very large number of connections. Diffusion convolutional neural networks (Atwood & Towsley 2016) use powers of the normalized adjacency matrix to blend features across different scales, sum these, pointwise multiply by weights, and pass through an activation function to create the node embeddings. Gilmer et al. (2017) introduced *message passing neural networks*, which defined convolutions on the graph as propagating messages from spatial neighbors. The "aggregate and combine" formulation of *GraphSAGE* (Hamilton et al. 2017a) fits into this framework.

Aggregate and combine: Graph convolutional networks (Kipf & Welling 2016a) take a weighted average of the neighbors and current node and then apply a linear mapping and

RELU. GraphSAGE (Hamilton et al. 2017a) applies a neural network layer to each neighbor and then takes the elementwise maximum to aggregate. Chiang et al. (2019) propose *diagonal enhancement* in which the previous embedding is weighted more than the neighbors. Kipf & Welling (2016a) introduced Kipf normalization, which normalizes the sum of the neighboring embeddings based on the degrees of the current node and its neighbors (see equation 13.19).

The *mixture model network* or *MoNet* (Monti et al. 2017) takes this one step further by *learning* a weighting based on these two quantities. They associate a pseudo-coordinate system with each node, where the positions of the neighbors depend on the degrees of the current node and the neighbor. They then learn a continuous function based on a mixture of Gaussians and sample this at the pseudo-coordinates of the neighbors to get the weights. In this way, they can learn the weightings for nodes and neighbors with arbitrary degrees. Pham et al. (2017) use a linear interpolation of the node embedding and neighbors with a different weighted combination for each dimension. The weight of this gating mechanism is generated as a function of the data.

Higher-order convolutional layers: Zhou & Li (2017) used higher-order convolutions by replacing the adjacency matrix $\mathbf{A}$ with $\tilde{\mathbf{A}} = \mathbf{Min}[\mathbf{A}^L + \mathbf{I}, \mathbf{1}]$ in the computations. In other words, the updates now sum together contributions from any nodes where there is at least one walk of length L. Abu-El-Haija et al. (2019) proposed MixHop, which computes node updates from the neighbors (using the adjacency matrix $\mathbf{A}$), the neighbors of the neighbors (using $\mathbf{A}^2$), and so on. They concatenate together these updates at each layer. Lee et al. (2018) combined information from nodes beyond the immediate neighbors using geometric *motifs*, which are small local geometric patterns in the graph (e.g., a fully connected clique of five nodes).

Residual connections: Kipf & Welling (2016a) proposed a residual connection in which the original embeddings are added to the updated ones. Hamilton et al. (2017b) concatenate the previous embedding to the output of the next layer (see equation 13.16). Rossi et al. (2020) present an inception-style network, where the node embedding is concatenated to not only the aggregation of its neighbors but also the aggregation of all neighbors within a walk of two (via computing powers of the adjacency matrix). Xu et al. (2018) introduced *jump knowledge connections* in which the final output at each node consists of the concatenated node embeddings throughout the network. Zhang & Meng (2019) present a general formulation of residual embeddings called *GResNet* and investigate several variations, in which the embeddings from the previous layer are added, the input embeddings are added, or versions of these that aggregate information from their neighbors (without further transformation) are added.

Attention in graph neural networks: Veličković et al. (2017) developed the graph attention network (figure 13.12c). Their formulation uses multiple heads whose outputs are combined together symmetrically. *Gated Attention Networks* (Zhang et al. 2018a) weight the output of the different heads in a way that depends on the data itself. Graph-BERT (Zhang et al. 2020) performs node classification using self-attention alone; the structure of the graph is captured by adding position embeddings to the data in a similar way to how the absolute or relative position of words is captured in the transformer (chapter 12). For example, they add positional information that depends on the number of hops between nodes in the graph.

Permutation invariance: In *DeepSets*, Zaheer et al. (2017) presented a general permutation invariant operator for processing sets. Janossy pooling (Murphy et al. 2018) accepts that many functions are not permutation equivariant, and instead uses a permutation-sensitive function and averages the results across many permutations.

Edge graphs: The notation of the *edge graph*, *line graph*, or *adjoint graph* dates to Whitney (1932). The idea of "weaving" layers that update node embeddings from node embeddings, node embeddings from edge embeddings, edge embeddings from edge embeddings, and edge embeddings from node embeddings was proposed by Kearnes et al. (2016), although here the

node-node and edge-edge updates do not involve the neighbors. Monti et al. (2018) introduced the dual-primal graph CNN, which is a modern formulation in a CNN framework that alternates between updating on the original graph and the edge graph.

Power of graph neural networks: Xu et al. (2019) argue that a neural network should be able to distinguish different graph structures; it is undesirable to map two graphs with the same initial node embeddings but different adjacency matrices to the same output. They identified graph structures that could not be distinguished by previous approaches such as GCNs (Kipf & Welling 2016a) and GraphSAGE (Hamilton et al. 2017a). They developed a more powerful architecture that has the same discriminative power as the Weifeiler-Lehman graph isomorphism test (Weisfeiler & Leman 1968), which is known to discriminate a broad class of graphs. This resulting *graph isomorphism network* was based on the aggregation operation:

$$\mathbf{h}_{k+1}^{(n)} = \mathbf{mlp}\left[(1+\epsilon_k)\,\mathbf{h}_{k+1}^{(n)} + \sum_{m\in \mathrm{ne}[n]} \mathbf{h}_m\right]. \tag{13.26}$$

Batches: The original paper on graph convolutional networks (Kipf & Welling 2016a) used full-batch gradient descent. This has memory requirements that are proportional to the number of nodes, embedding size, and number of layers during training. Since then, three types of method have been proposed to reduce the memory requirements, and create batches for SGD in the transductive setting: node sampling, layer sampling, and sub-graph sampling.

Node sampling methods start by randomly selecting a subset of target nodes and then work back through the network, adding a subset of the nodes in the receptive field at each stage. GraphSAGE (Hamilton et al. 2017a) proposed a fixed number of neighborhood samples as in figure 13.10b. Chen et al. (2017) introduce a variance reduction technique but this uses historical activations of nodes, and so still has a high memory requirement. *PinSAGE* (Ying et al. 2018a) used random walks from the target nodes and chooses the top K nodes with the highest visit count. This prioritizes ancestors that are more closely connected.

Node sampling methods still require increasing numbers of nodes as we pass back through the graph. *Layer sampling methods* address this by directly sampling the receptive field in each layer independently. Examples of layer sampling include FastGCN (Chen et al. 2018a), adaptive sampling (Huang et al. 2018), and layer-dependent importance sampling (Zou et al. 2019).

Subgraph sampling methods randomly draw subgraphs or divide the original graph into subgraphs. These are then trained as independent data examples. Examples of these approaches include GraphSAINT (Zeng et al. 2019a), which samples sub-graphs during training using random walks, and then runs a full GCN on the subgraph, while also correcting for the bias and variance of the minibatch. Cluster GCN(Chiang et al. 2019) partitions the graph into clusters (by maximizing the embedding utilization or number of within-batch edges) in a pre-processing stage and randomly selects clusters to form minibatches. To create more randomness, they train random subsets of these clusters plus the edges between them (see figure 13.11).

Wolfe et al. (2021) propose a method for distributed training, which both partitions the graph and trains narrower GCNs in parallel by partitioning the feature space at different layers. More information about sampling graphs can be found in Rozemberczki et al. (2020).

Regularization and normalization: Rong et al. (2020) proposed *DropEdge*, which randomly drops edges from the graph during each iteration of training by masking the adjacency matrix. This can be done for the whole neural network, or differently in each layer (layer-wise DropEdge). In a sense, this is similar to dropout in that it breaks connections in the flow of data, but it can also be considered an augmentation method since changing the graph is similar to perturbing the data. Schlichtkrull et al. (2018), Teru et al. (2020) and Veličković et al. (2017) also propose

randomly dropping edges from the graph as a form of regularization similar to dropout. Node sampling methods (Hamilton et al. 2017a; Huang et al. 2018; Chen et al. 2018a) can also be thought of as regularizers. Hasanzadeh et al. (2020) present a general framework called *DropConnect* that unifies many of the above approaches.

There are also many proposed normalization schemes for graph neural networks, including *PairNorm* (Zhao & Akoglu 2019), weight normalization (Oono & Suzuki 2019), differentiable group normalization (Zhou et al. 2020b), and GraphNorm (Cai et al. 2021).

Multi-relational graphs: Schlichtkrull et al. (2018) proposed a variation of graph convolutional networks for multi-relational graphs (i.e., graphs with more than one edge type). Their scheme separately aggregates information from each edge type, using different parameters. If there are many edge types, then the number of parameters may become large, and to combat this they propose that each edge type uses a different weighting of a basis set of parameters.

Hierarchical representations and pooling: CNNs for image classification gradually decrease the size of the representation, but increase the number of channels as the network progresses. However, the GCNs for graph classification in this chapter maintain the entire graph until the last layer and then combine all of the nodes to compute the final prediction. Ying et al. (2018b) propose *DiffPool*, which clusters graph nodes to make a graph that gets progressively smaller as the depth increases in a way that is differentiable, so can be learned. This can be done based on the graph structure alone, or adaptively based on the graph structure and the embeddings. Other pooling methods include SortPool (Zhang et al. 2018b) and self-attention graph-pooling (Lee et al. 2019). A comparison of pooling layers for graph neural networks can be found in Grattarola et al. (2022). Gao & Ji (2019) propose an encoder-decoder structure for graphs based on the U-Net (see figure 11.10).

Geometric graphs: MoNet (Monti et al. 2017) can be easily adapted to exploit geometric information because neighboring nodes have well-defined spatial positions. They learn a mixture of Gaussians function and sample from this based on the relative coordinates of the neighbor. In this way, they can weight neighboring nodes based on their relative positions as in standard convolutional neural networks even though these positions are not consistent. The geodesic CNN (Masci et al. 2015) and anisotropic CNN (Boscaini et al. 2016) both adapt convolution to manifolds (i.e., surfaces) as represented by triangular meshes. They locally approximate the surface as a plane and define a coordinate system on this plane around the current node.

Oversmoothing and suspended animation: Unlike other deep learning models, graph neural networks did not until recently benefit significantly from increasing depth. Indeed, the original GCN paper (Kipf & Welling 2016a) and GraphSAGE (Hamilton et al. 2017a) both only use two layers, and Chiang et al. (2019) train a five-layer Cluster-GCN to get state-of-the-art performance on the PPI dataset. One possible explanation is *over-smoothing* (Li et al. 2018b); at each layer, the network incorporates information from a larger neighborhood and it may be that this ultimately results in the dissolution of (important) local information. Indeed (Xu et al. 2018) prove that the influence of one node on another is proportional to the probability of reaching that node in a K-step random walk, and this approaches the stationary distribution of walks over the graph with increasing K causing the local neighborhood to be washed out.

Alon & Yahav (2020) proposed a different explanation for why performance doesn't improve with network depth. They argue that adding depth allows information to be aggregated from longer paths. However, in practice, the exponential growth in the number of neighbors means there is a bottleneck whereby too much information is "squashed" into the fixed-size node embeddings.

Ying et al. (2018a) also note that when the depth of the network exceeds a certain limit, the gradients no longer propagate back, and learning fails for both the training and test data. They

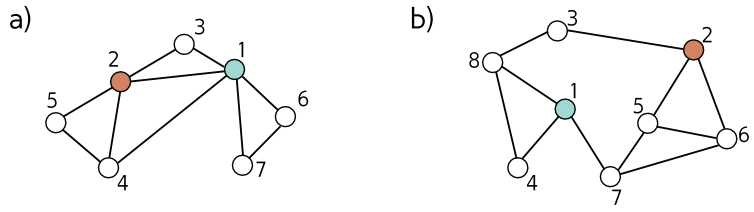

Figure 13.14 Graphs for problems 13.1, 13.3, and 13.8.

term this effect *suspended animation*. This is very similar to what happens when many layers are naïvely added to convolutional neural networks (figure 11.2). They propose a family of residual connections that allow deeper networks to be trained. Vanishing gradients (section 7.3) have also been identified as a limitation by Li et al. (2021b).

It has recently become possible train to deeper graph neural networks using various forms of residual connection (Xu et al. 2018; Li et al. 2020a; Gong et al. 2020; Chen et al. 2020b; Xu et al. 2021a). Li et al. (2021a) train a state-of-the-art model with more than 1000 layers using an invertible network to reduce the memory requirements of training (see chapter 16).

Problems

Problem 13.1 Write out the adjacency matrices for the two graphs in figure 13.14.

Problem 13.2 Draw graphs that correspond to the following adjacency matrices:

$$\mathbf{A}_1 = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \end{bmatrix} \quad \text{and} \quad \mathbf{A}_2 = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}.$$

Problem 13.3 Consider the two graphs in figure 13.14. How many ways are there to walk from node one to node two in (i) three steps and (ii) seven steps?

Problem 13.4 The diagonal of $\mathbf{A}^2$ in figure 13.4c contains the number of edges that connect to each corresponding node. Explain this phenomenon.

Problem 13.5 What permutation matrix is responsible for the transformation between the graphs in figures 13.5a–c and figure 13.5d–f?

Problem 13.6 Prove that:

$$\text{sig}\left[\beta_K + \boldsymbol{\omega}_K \mathbf{H}_K \mathbf{1}\right] = \text{sig}\left[\beta_K + \boldsymbol{\omega}_K \mathbf{H}_K \mathbf{P} \mathbf{1}\right], \tag{13.27}$$

where $\mathbf{P}$ is an $N \times N$ permutation matrix (a matrix which is all zeros except for exactly one entry in each row and each column which is one) and $\mathbf{1}$ is an $N \times 1$ vector of ones.

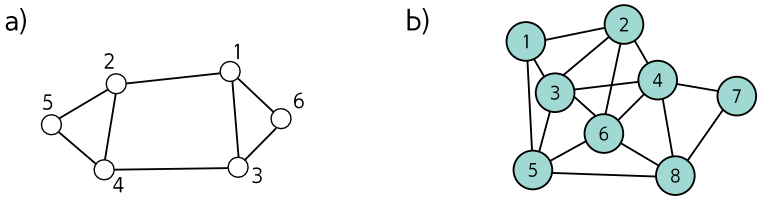

Figure 13.15 Graphs for problems 13.11–13.13.

Problem 13.7 Consider the simple GNN layer:

$$\begin{aligned}\mathbf{H}_{k+1} &= \text{GraphLayer}[\mathbf{H}_k, \mathbf{A}] \\ &= \mathbf{a}\left[\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k[\mathbf{H}_k; \mathbf{H}_k\mathbf{A}]\right],\end{aligned} \tag{13.28}$$

where $\mathbf{H}$ is a $D \times N$ matrix containing the N node embeddings in its columns, $\mathbf{A}$ is the $N \times N$ adjacency matrix, $\boldsymbol{\beta}$ is the bias vector and $\boldsymbol{\Omega}$ is the weight matrix. The notation $[\mathbf{H}_k; \mathbf{H}_k\mathbf{A}]$ indicates vertical concatenation of the matrices $\mathbf{H}_k$ and $\mathbf{H}_k\mathbf{A}$. Show that this layer is equivariant to permutations of the node order, so that:

$$\text{GraphLayer}[\mathbf{H}_k, \mathbf{A}]\mathbf{P} = \text{GraphLayer}[\mathbf{H}_k\mathbf{P}, \mathbf{P}^T\mathbf{A}\mathbf{P}], \tag{13.29}$$

where $\mathbf{P}$ is an $N \times N$ permutation matrix.

Appendix C.1.8 Permutation matrix

Problem 13.8 What is the degree matrix $\mathbf{D}$ for each of the graphs in figure 13.14?

Problem 13.9 The authors of GraphSAGE (Hamilton et al. 2017a) propose a pooling method, in which the node embedding is averaged together with its neighbors so that:

$$\mathbf{agg}[n] = \frac{1}{1+|\text{ne}[n]|}\left(\mathbf{h}_n + \sum_{m \in \text{ne}[n]} \mathbf{h}_m\right). \tag{13.30}$$

Show how this operation can be computed simultaneously for all of the node embeddings in the $D \times N$ embedding matrix $\mathbf{H}$ using linear algebra. You will need to use both the adjacency matrix $\mathbf{A}$ and the degree matrix $\mathbf{D}$.

Problem 13.10 Devise a graph attention mechanism based on dot-product self-attention and draw its mechanism in the style of figure 13.12.

Problem 13.11 Draw the edge graph associated with the graph in figure 13.15a.

Problem 13.12 Draw the node graph that corresponds to the edge graph in figure 13.15b.

Problem 13.13 For a general undirected graph, describe how the adjacency matrix of the node graph relates to the adjacency matrix of the corresponding edge graph.

Problem 13.14 Design a layer that updates a node embedding $\mathbf{h}_n$ based on its neighboring node embeddings $\{\mathbf{h}_m\}_{m \in \text{ne}[n]}$ and neighboring edge embeddings $\{\mathbf{e}_m\}_{m \in \text{nee}[n]}$. You should consider the possibility that the edge embeddings are not the same size as the node embeddings.

Chapter 14

Generative adversarial networks

Chapter 15

Variational autoencoders

Chapter 16

Normalizing flows

Chapter 17

Diffusion models

Chapter 18

Reinforcement learning

Chapter 19

Why does deep learning work?

Appendix A

Notation

This is a brief guide to the notational conventions used in this text.

Appendix B

Probability

Appendix C

Maths

Bibliography

Abdel-Hamid, O., Mohamed, A.-r., Jiang, H., & Penn, G. (2012) Applying convolutional neural networks concepts to hybrid nn-hmm model for speech recognition. In *2012 IEEE international conference on Acoustics, speech and signal processing (ICASSP)*, pp. 4277–4280. IEEE. 182

Abesser, J., Mimilakis, S. I., Gräfe, R., Lukashevich, H., & Fraunhofer, I. (2017) Acoustic scene classification by combining autoencoder-based dimensionality reduction and convolutional neural networks. In *Proc. of the Detection and Classification of Acoustic Scenes and Events 2017 Workshop (DCASE2017)*, pp. 7–11. 157

Abu-El-Haija, S., Perozzi, B., Kapoor, A., Alipourfard, N., Lerman, K., Harutyunyan, H., Ver Steeg, G., & Galstyan, A. (2019) Mixhop: Higher-order graph convolutional architectures via sparsified neighborhood mixing. In *international conference on machine learning*, pp. 21–29. PMLR. 266

Aggarwal, C. C., Hinneburg, A., & Keim, D. A. (2001) On the surprising behavior of distance metrics in high dimensional space. In *International conference on database theory*, pp. 420–434. Springer. 132

Aiken, M., & Park, M. (2010) The efficacy of round-trip translation for mt evaluation. *Translation Journal* **14** (1): 1–10. 158

Ainslie, J., Ontanon, S., Alberti, C., Cvicek, V., Fisher, Z., Pham, P., Ravula, A., Sanghai, S., Wang, Q., & Yang, L. (2020) Etc: Encoding long and structured inputs in transformers. *arXiv preprint arXiv:2004.08483* . 239

Alhashim, I., & Wonka, P. (2018) High quality monocular depth estimation via transfer learning. *arXiv preprint arXiv:1812.11941* . 205

Ali, A., Touvron, H., Caron, M., Bojanowski, P., Douze, M., Joulin, A., Laptev, I., Neverova, N., Synnaeve, G., Verbeek, J., & others. (2021) Xcit: Cross-covariance image transformers. *Advances in neural information processing systems* **34**: 20014–20027. 241

Alon, U., & Yahav, E. (2020) On the bottleneck of graph neural networks and its practical implications. *arXiv preprint arXiv:2006.05205* . 268

An, G. (1996) The effects of adding noise during backpropagation training on a generalization performance. *Neural computation* **8** (3): 643–674. 156

Arnab, A., Dehghani, M., Heigold, G., Sun, C., Lučić, M., & Schmid, C. (2021) Vivit: A video vision transformer. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 6836–6846. 241

Arora, R., Basu, A., Mianjy, P., & Mukherjee, A. (2016) Understanding deep neural networks with rectified linear units. *arXiv preprint arXiv:1611.01491* . 49

Arora, S., Li, Z., & Lyu, K. (2018) Theoretical analysis of auto rate-tuning by batch normalization. *arXiv preprint arXiv:1812.03981* . 204

Atwood, J., & Towsley, D. (2016) Diffusion-convolutional neural networks. *Advances in neural information processing systems* **29**. 265

Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016) Layer normalization. *arXiv preprint arXiv:1607.06450* . 203

Bachlechner, T., Majumder, B. P., Mao, H., Cottrell, G., & McAuley, J. (2021) Rezero is all you need: Fast convergence at large depth. In *Uncertainty in Artificial Intelligence*, pp. 1352–1361. PMLR. 240

Bahdanau, D., Cho, K., & Bengio, Y. (2014) Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* . 235, 237

Balduzzi, D., Frean, M., Leary, L., Lewis, J., Ma, K. W.-D., & McWilliams, B. (2017)

The shattered gradients problem: If resnets are the answer, then what is the question? In *International Conference on Machine Learning*, pp. 342–350. PMLR. 187, 201, 203, 204

Barber, D., & Bishop, C. (1997) Ensemble learning for multi-layer networks. *Advances in neural information processing systems* **10**. 156

Barrett, D. G. T., & Dherin, B., (2021) Implicit gradient regularization. 155

Barron, J. T., (2019) A general and adaptive robust loss function. 70

Bartlett, P. L., Foster, D. J., & Telgarsky, M. J. (2017a) Spectrally-normalized margin bounds for neural networks. In *Advances in Neural Information Processing Systems*, pp. 6240–6249. 154

Bartlett, P. L., Harvey, N., Liaw, C., & Mehrabian, A., (2017b) Nearly-tight vc-dimension and pseudodimension bounds for piecewise linear neural networks. 131

Bau, D., Zhou, B., Khosla, A., Oliva, A., & Torralba, A. (2017) Network dissection: Quantifying interpretability of deep visual representations. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 6541–6549. 183

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M., (2018) Automatic differentiation in machine learning: a survey. 109

Bayer, M., Kaufhold, M.-A., & Reuter, C. (2021) A survey on data augmentation for text classification. *arXiv preprint arXiv:2107.03158* . 158

Belinkov, Y., & Bisk, Y. (2017) Synthetic and natural noise both break neural machine translation. *arXiv preprint arXiv:1711.02173* . 157

Belkin, M., Hsu, D., Ma, S., & Mandal, S. (2019) Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences* **116** (32): 15849–15854. 127, 131

Beltagy, I., Peters, M. E., & Cohan, A. (2020) Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150* . 239

Bender, E. M., & Koller, A. (2020) Climbing towards nlu: On meaning, form, and understanding in the age of data. In *Proceedings of the 58th annual meeting of the association for computational linguistics*, pp. 5185–5198. 237

Bergstra, J., & Bengio, Y. (2012) Random search for hyper-parameter optimization. *Journal of machine learning research* **13** (2). 133

Bergstra, J. S., Bardenet, R., Bengio, Y., & Kégl, B. (2011) Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pp. 2546–2554. 133

Bertasius, G., Wang, H., & Torresani, L. (2021) Is space-time attention all you need for video understanding? In *ICML*, volume 2, p. 4. 241

Beyer, K., Goldstein, J., Ramakrishnan, R., & Shaft, U. (1999) When is "nearest neighbor" meaningful? In *International conference on database theory*, pp. 217–235. Springer. 132

Bishop, C. M. (1994) Mixture density networks. 70

Bishop, C. M. (1995) Regularization and complexity control in feed-forward networks. 155, 156

Bishop, C. M. (2006) *Pattern recognition and machine learning*. Springer. 156

Bjorck, N., Gomes, C. P., Selman, B., & Weinberger, K. Q. (2018) Understanding batch normalization. *Advances in neural information processing systems* **31**. 204

Blundell, C., Cornebise, J., Kavukcuoglu, K., & Wierstra, D. (2015) Weight uncertainty in neural network. In *International conference on machine learning*, pp. 1613–1622. PMLR. 156

Boscaini, D., Masci, J., Rodolà, E., & Bronstein, M. (2016) Learning shape correspondence with anisotropic convolutional neural networks. *Advances in neural information processing systems* **29**. 268

Bottou, L. (2012) *Stochastic Gradient Descent Tricks*, pp. 421–436. Springer Berlin Heidelberg. 90

Bottou, L., Curtis, F. E., & Nocedal, J., (2018) Optimization methods for large-scale machine learning. 89

Bottou, L., Soulié, F. F., Blanchet, P., & Liénard, J.-S. (1990) Speaker-independent isolated digit recognition: Multilayer perceptrons vs. dynamic time warping. *Neural Networks* **3** (4): 453–465. 180

Bousselham, W., Thibault, G., Pagano, L., Machireddy, A., Gray, J., Chang, Y. H., & Song, X. (2021) Efficient self-ensemble framework for semantic segmentation. *arXiv preprint arXiv:2111.13280* . 160

Bowman, S. R., & Dahl, G. E. (2021) What will it take to fix benchmarking in natural language understanding? *arXiv preprint arXiv:2104.02145* . 236

Bromley, J., Guyon, I., LeCun, Y., Säckinger, E., & Shah, R. (1993) Signature verification using a" siamese" time delay neural network. *Advances in neural information processing systems* **6**. 180

Bronstein, M. M., Bruna, J., Cohen, T., & Veličković, P. (2021) Geometric deep learning: Grids, groups, graphs, geodesics, and gauges. *arXiv preprint arXiv:2104.13478* . 265

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., & others. (2020) Language models are few-shot learners. *Advances in neural information processing systems* **33**: 1877–1901. 157, 236, 239

Bruna, J., Zaremba, W., Szlam, A., & LeCun, Y. (2013) Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203* . 265

Bryson, A., Ho, Y.-C., & Siouris, G. (1979) Applied optimal control: Optimization, estimation, and control. *Systems, Man and Cybernetics, IEEE Transactions on* **9**: 366 – 367. 108

Bubeck, S., & Sellke, M., (2021) A universal law of robustness via isoperimetry. 132

Burda, Y., Grosse, R., & Salakhutdinov, R., (2016) Importance weighted autoencoders. 70

Buschjäger, S., & Morik, K., (2021) There is no double-descent in random forests. 131

Cai, T., Luo, S., Xu, K., He, D., Liu, T.-y., & Wang, L. (2021) Graphnorm: A principled approach to accelerating graph neural network training. In *International Conference on Machine Learning*, pp. 1204–1215. PMLR. 268

Calimeri, F., Marzullo, A., Stamile, C., & Terracina, G. (2017) Biomedical data augmentation using adversarial neural networks. In *International conference on artificial neural networks*, pp. 626–634. Springer. 157

Cao, Z., Qin, T., Liu, T.-Y., Tsai, M.-F., & Li, H. (2007) Learning to rank: from pairwise approach to listwise approach. In *Proceedings of the 24th international conference on Machine learning*, pp. 129–136. 70

Carion, N., Massa, F., Synnaeve, G., Usunier, N., Kirillov, A., & Zagoruyko, S. (2020) End-to-end object detection with transformers. In *European conference on computer vision*, pp. 213–229. Springer. 240

Cauchy, A. (1847) Methode generale pour la resolution des systemes d'equations simultanees. *C.R. Acad. Sci. Paris* **25**: 536–538. 89

Chami, I., Abu-El-Haija, S., Perozzi, B., Ré, C., & Murphy, K. (2020) Machine learning on graphs: A model and comprehensive taxonomy. *arXiv preprint arXiv:2005.03675* p. 1. 265

Chang, Y.-L., Liu, Z. Y., Lee, K.-Y., & Hsu, W. (2019) Free-form video inpainting with 3d gated convolution and temporal patchgan. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 9066–9075. 182

Chaudhari, P., Choromanska, A., Soatto, S., LeCun, Y., Baldassi, C., Borgs, C., Chayes, J., Sagun, L., & Zecchina, R., (2017) Entropy-sgd: Biasing gradient descent into wide valleys. 156

Chen, H., Wang, Y., Guo, T., Xu, C., Deng, Y., Liu, Z., Ma, S., Xu, C., Xu, C., & Gao, W. (2021) Pre-trained image processing transformer. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 12299–12310. 240

Chen, J., Ma, T., & Xiao, C. (2018a) Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247* . 267, 268

Chen, J., Zhu, J., & Song, L. (2017) Stochastic training of graph convolutional networks with variance reduction. *arXiv preprint arXiv:1710.10568* . 267

Chen, L.-C., Papandreou, G., Kokkinos, I., Murphy, K., & Yuille, A. L. (2014a) Semantic image segmentation with deep convolutional nets and fully connected crfs. *arXiv preprint arXiv:1412.7062* . 181

Chen, M., Radford, A., Child, R., Wu, J., Jun, H., Luan, D., & Sutskever, I. (2020a) Generative pretraining from pixels. In *International conference on machine learning*, pp. 1691–1703. PMLR. 240, 241

Chen, M., Wei, Z., Huang, Z., Ding, B., & Li, Y. (2020b) Simple and deep graph convolutional networks. In *International Conference on Machine Learning*, pp. 1725–1735. PMLR. 269

Chen, R. T., Rubanova, Y., Bettencourt, J., & Duvenaud, D. K. (2018b) Neural ordinary differential equations. *Advances in neural information processing systems* **31**. 202

Chen, T., Fox, E., & Guestrin, C. (2014b) Stochastic gradient hamiltonian monte carlo. In *International conference on machine learning*, pp. 1683–1691. PMLR. 156

Chen, T., Kornblith, S., Norouzi, M., & Hinton, G. (2020c) A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pp. 1597–1607. PMLR. 157

Chen, T., Xu, B., Zhang, C., & Guestrin, C., (2016) Training deep nets with sublinear memory cost. 109

Chen, W., Liu, T.-Y., Lan, Y., Ma, Z.-M., & Li, H. (2009) Ranking measures and loss functions in learning to rank. *Advances in Neural Information Processing Systems* **22**: 315–323. 70

Chen, Y.-C., Li, L., Yu, L., El Kholy, A., Ahmed, F., Gan, Z., Cheng, Y., & Liu, J. (2020d) Uniter: Universal image-text representation learning. In *European conference on computer vision*, pp. 104–120. Springer. 240

Chiang, W.-L., Liu, X., Si, S., Li, Y., Bengio, S., & Hsieh, C.-J. (2019) Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 257–266. 266, 267, 268

Child, R., Gray, S., Radford, A., & Sutskever, I. (2019) Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509* . 239

Cho, K., Van Merriënboer, B., Bahdanau, D., & Bengio, Y. (2014) On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259* . 235

Choi, D., Shallue, C. J., Nado, Z., Lee, J., Maddison, C. J., & Dahl, G. E. (2019) On empirical comparisons of optimizers for deep learning. *arXiv preprint arXiv:1910.05446* . 92

Choromanski, K., Likhosherstov, V., Dohan, D., Song, X., Gane, A., Sarlos, T., Hawkins, P., Davis, J., Mohiuddin, A., Kaiser, L., & others. (2020) Rethinking attention with performers. *arXiv preprint arXiv:2009.14794* . 238, 240

Chorowski, J., & Jaitly, N. (2016) Towards better decoding and language model integration in sequence to sequence models. *arXiv preprint arXiv:1612.02695* . 156

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., & others. (2022) Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311* . 236

Chu, X., Tian, Z., Wang, Y., Zhang, B., Ren, H., Wei, X., Xia, H., & Shen, C. (2021) Twins: Revisiting the design of spatial attention in vision transformers. *Advances in Neural Information Processing Systems* **34**: 9355–9366. 241

Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014) Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555* . 235

Çiçek, Ö., Abdulkadir, A., Lienkamp, S. S., Brox, T., & Ronneberger, O. (2016) 3d u-net: learning dense volumetric segmentation from sparse annotation. In *International conference on medical image computing and computer-assisted intervention*, pp. 424–432. Springer. 205

Clark, M. (2022) The engineer who claimed a google ai is sentient has been fired. 237

Clevert, D.-A., Unterthiner, T., & Hochreiter, S. (2015) Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289* . 32

Cohen, N., Sharir, O., & Shashua, A. (2015) On the expressive power of deep learning: A tensor analysis. *CoRR* **abs/1509.05009**. 49

Cohen, T., & Welling, M. (2016) Group equivariant convolutional networks. In *International conference on machine learning*, pp. 2990–2999. PMLR. 182

Conneau, A., Schwenk, H., Barrault, L., & Lecun, Y. (2016) Very deep convolutional networks for text classification. *arXiv preprint arXiv:1606.01781* . 182

Cordonnier, J.-B., Loukas, A., & Jaggi, M. (2019) On the relationship between self-attention and convolutional layers. *arXiv preprint arXiv:1911.03584* . 238

Cordts, M., Omran, M., Ramos, S., Rehfeld, T., Enzweiler, M., Benenson, R., Franke, U., Roth, S., & Schiele, B. (2016) The cityscapes dataset for semantic urban scene understanding. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3213–3223. 150

Coulombe, C. (2018) Text data augmentation made simple by leveraging nlp cloud apis. *arXiv preprint arXiv:1812.04718* . 157

Cristianini, M., & Shawe-Taylor, J. (2000) *An introduction to support vector machines*. Cambridge University Press. 71

Cybenko, G. (1989) Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems* **2** (4): 303–314. 32

Dai, H., Dai, B., & Song, L. (2016) Discriminative embeddings of latent variable models for structured data. In *International conference on machine learning*, pp. 2702–2711. PMLR. 265

Dai, J., Qi, H., Xiong, Y., Li, Y., Zhang, G., Hu, H., & Wei, Y. (2017) Deformable convolutional networks. In *Proceedings of the IEEE international conference on computer vision*, pp. 764–773. 183

Daigavane, A., Balaraman, R., & Aggarwal, G. (2021) Understanding convolutions on graphs. *Distill* . 264

Daniluk, M., Rocktäschel, T., Welbl, J., & Riedel, S. (2017) Frustratingly short attention spans in neural language modeling. *arXiv preprint arXiv:1702.04521* . 237

Dar, Y., Muthukumar, V., & Baraniuk, R. G., (2021) A farewell to the bias-variance tradeoff? an overview of the theory of overparameterized machine learning. 132

De, S., & Smith, S. (2020) Batch normalization biases residual blocks towards the identity function in deep networks. *Advances in Neural Information Processing Systems* **33**: 19964–19975. 205

Dechter, R. (1986) Learning while searching in constraint-satisfaction-problems. In *Proceedings of the Fifth AAAI National Conference on Artificial Intelligence*, AAAI'86, p. 178–183. AAAI Press. 48

Defferrard, M., Bresson, X., & Vandergheynst, P. (2016) Convolutional neural networks on graphs with fast localized spectral filtering. *Advances in neural information processing systems* **29**. 265

Dehghani, M., Tay, Y., Gritsenko, A. A., Zhao, Z., Houlsby, N., Diaz, F., Metzler, D., & Vinyals, O. (2021) The benchmark lottery. *arXiv preprint arXiv:2107.07002* . 236

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2018) Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* . 157, 236

DeVries, T., & Taylor, G. W. (2017a) Dataset augmentation in feature space. *arXiv preprint arXiv:1702.05538* . 156

DeVries, T., & Taylor, G. W. (2017b) Improved regularization of convolutional neural networks with cutout. *arXiv preprint arXiv:1708.04552* . 183

Ding, M., Xiao, B., Codella, N., Luo, P., Wang, J., & Yuan, L. (2022) Davit: Dual attention vision transformers. *arXiv preprint arXiv:2204.03645* . 241

Doersch, C., Gupta, A., & Efros, A. A. (2015) Unsupervised visual representation learning by context prediction. In *Proceedings of the IEEE international conference on computer vision*, pp. 1422–1430. 157

Domingos, P. (2000) A unified bias-variance decomposition. In *Proceedings of 17th international conference on machine learning*, pp. 231–238. Morgan Kaufmann Stanford. 130

Domke, J., (2010) Statistical machine learning. https://people.cs.umass.edu/ domke/. 112

Dong, X., Bao, J., Chen, D., Zhang, W., Yu, N., Yuan, L., Chen, D., & Guo, B. (2022) Cswin transformer: A general vision transformer backbone with cross-shaped windows. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 12124–12134. 241

Dorta, G., Vicente, S., Agapito, L., Campbell, N. D. F., & Simpson, I., (2018) Structured uncertainty prediction networks. 70

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., & others. (2020) An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* . 235, 240, 241

Dozat, T. (2016) Incorporating nesterov momentum into adam. 91

Du, N., Huang, Y., Dai, A. M., Tong, S., Lepikhin, D., Xu, Y., Krikun, M., Zhou, Y., Yu, A. W., Firat, O., & others. (2022) Glam: Efficient scaling of language models with mixture-of-experts. In *International Conference on Machine Learning*, pp. 5547–5569. PMLR. 236

Duchi, J., Hazan, E., & Singer, Y. (2011) Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research* **12** (Jul): 2121–2159. 91

Dufter, P., Schmitt, M., & Schütze, H. (2021) Position information in transformers: An overview. *Computational Linguistics* pp. 1–31. 239

Dumoulin, V., & Visin, F. (2016) A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285* . 180

Dupont, E., Doucet, A., & Teh, Y. W. (2019) Augmented neural odes. *Advances in Neural Information Processing Systems* **32**. 202

Duvenaud, D. K., Maclaurin, D., Iparraguirre, J., Bombarell, R., Hirzel, T., Aspuru-Guzik, A., & Adams, R. P. (2015) Convolutional networks on graphs for learning molecular fingerprints. *Advances in neural information processing systems* **28**. 265

Ebrahimi, J., Rao, A., Lowd, D., & Dou, D. (2017) Hotflip: White-box adversarial examples for text classification. *arXiv preprint arXiv:1712.06751* . 157

El Asri, L., & Prince, S. (2020) Tutorial #6: neural natural language generation – decoding algorithms. 237

Eldan, R., & Shamir, O. (2015) The power of depth for feedforward neural networks. *CoRR* **abs/1512.03965**. 49

Eren, L., Ince, T., & Kiranyaz, S. (2019) A generic intelligent bearing fault diagnosis system using compact adaptive 1d cnn classifier. *Journal of Signal Processing Systems* **91** (2): 179–189. 182

Erhan, D., Bengio, Y., Courville, A., & Vincent, P. (2009) Visualizing higher-layer features of a deep network. *University of Montreal* **1341** (3): 1. 183

Errica, F., Podda, M., Bacciu, D., & Micheli, A. (2019) A fair comparison of graph neural networks for graph classification. *arXiv preprint arXiv:1912.09893* . 265

Esteves, C., Allen-Blanchette, C., Zhou, X., & Daniilidis, K. (2017) Polar transformer networks. *arXiv preprint arXiv:1709.01889* . 182

Falk, T., Mai, D., Bensch, R., Çiçek, Ö., Abdulkadir, A., Marrakchi, Y., Böhm, A., Deubner, J., Jäckel, Z., Seiwald, K., & others. (2019) U-net: deep learning for cell counting, detection, and morphometry. *Nature methods* **16** (1): 67–70. 198

Falkner, S., Klein, A., & Hutter, F. (2018) BOHB: robust and efficient hyperparameter optimization at scale. *CoRR* **abs/1807.01774**. 133

Fallah, N., Gu, H., Mohammad, K., Seyyedsalehi, S. A., Nourijelyani, K., & Eshraghian, M. R. (2009) Nonlinear poisson regression using neural networks: a simulation study. *Neural Computing and Applications* **18** (8): 939–943. 70

Fan, A., Lewis, M., & Dauphin, Y. (2018) Hierarchical neural story generation. *arXiv preprint arXiv:1805.04833* . 237

Fan, H., Xiong, B., Mangalam, K., Li, Y., Yan, Z., Malik, J., & Feichtenhofer, C. (2021) Multiscale vision transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 6824–6835. 241

Fan, K., Li, B., Wang, J., Zhang, S., Chen, B., Ge, N., & Yan, Z. (2020) Neural zero-inflated quality estimation model for automatic speech recognition system. In *Interspeech 2020, 21st Annual Conference of the International Speech Communication Association, Virtual Event, Shanghai, China, 25-29 October 2020*, ed. by H. Meng, B. Xu, & T. F. Zheng, pp. 606–610. ISCA. 70

Fang, Y., Liao, B., Wang, X., Fang, J., Qi, J., Wu, R., Niu, J., & Liu, W. (2021) You only look at one sequence: Rethinking transformer in vision through object detection. *Advances in Neural Information Processing Systems* **34**: 26183–26197. 240

Feng, S. Y., Gangal, V., Kang, D., Mitamura, T., & Hovy, E. (2020) Genaug: Data augmentation for finetuning text generators. *arXiv preprint arXiv:2010.01794* . 157

Fernández-Madrigal, J.-A., & González, J. (2002) Multihierarchical graph search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **24** (1): 103–113. 245

Fort, S., Hu, H., & Lakshminarayanan, B., (2020) Deep ensembles: A loss landscape perspective. 155

forthcoming. (2022) Big-bench collaboration. beyond the imitation game: Measuring and extrapolating the capabilities of language models. 236

Frankle, J., Dziugaite, G. K., Roy, D. M., & Carbin, M., (2020) Linear mode connectivity and the lottery ticket hypothesis. 155

Freund, Y., & Schapire, R. E. (1995) A decision-theoretic generalization of on-line learning and an application to boosting. In *Computational Learning Theory: Eurocolt '95*, pp. 23–37. 71

Friedman, J. H. (1997) On bias, variance, 0/1—loss, and the curse-of-dimensionality. *Data*

mining and knowledge discovery **1** (1): 55–77. 130

Fukushima, K. (1969) Visual feature extraction by a multilayered network of analog threshold elements. *IEEE Transactions on Systems Science and Cybernetics* **5** (4): 322–333. 31

Gal, Y., & Ghahramani, Z., (2015) Dropout as a bayesian approximation: Representing model uncertainty in deep learning. 156

Gales, M. J. (1998) Maximum likelihood linear transformations for hmm-based speech recognition. *Computer speech & language* **12** (2): 75–98. 157

Gales, M. J., Ragni, A., AlDamarki, H., & Gautier, C. (2009) Support vector machines for noise robust asr. In *2009 IEEE Workshop on Automatic Speech Recognition & Understanding*, pp. 205–210. IEEE. 157

Ganaie, M., Hu, M., & others. (2021) Ensemble deep learning: A review. *arXiv preprint arXiv:2104.02395* . 155

Gao, H., & Ji, S. (2019) Graph u-nets. In *international conference on machine learning*, pp. 2083–2092. PMLR. 268

Garipov, T., Izmailov, P., Podoprikhin, D., Vetrov, D. P., & Wilson, A. G. (2018) Loss surfaces, mode connectivity, and fast ensembling of dnns. *Advances in neural information processing systems* **31**. 155

Gastaldi, X. (2017a) Shake-shake regularization. *arXiv preprint arXiv:1705.07485* . 202

Gastaldi, X. (2017b) Shake-shake regularization of 3-branch residual networks. 202

Gidaris, S., Singh, P., & Komodakis, N. (2018) Unsupervised representation learning by predicting image rotations. *arXiv preprint arXiv:1803.07728* . 157

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., & Dahl, G. E. (2017) Neural message passing for quantum chemistry. In *International conference on machine learning*, pp. 1263–1272. PMLR. 265

Girdhar, R., Carreira, J., Doersch, C., & Zisserman, A. (2019) Video action transformer network. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 244–253. 240

Glorot, X., & Bengio, Y. (2010) Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, ed. by Y. W. Teh & M. Titterington, volume 9 of *Proceedings of Machine Learning Research*, pp. 249–256. PMLR. 109, 182

Glorot, X., Bordes, A., & Bengio, Y. (2011) Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 315–323. JMLR Workshop and Conference Proceedings. 31, 32

Goh, G. (2017) Why momentum really works. *Distill* . 90

Gomez, A. N., Ren, M., Urtasun, R., & Grosse, R. B. (2017) The reversible residual network: Backpropagation without storing activations. *Advances in neural information processing systems* **30**. 109

Gong, S., Bahri, M., Bronstein, M. M., & Zafeiriou, S. (2020) Geometrically principled connections in graph neural networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 11415–11424. 269

Goodfellow, I., Bengio, Y., & Courville, A. (2016) *Deep learning*. MIT press. 155

Goodfellow, I. J., Shlens, J., & Szegedy, C. (2014) Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* . 157

Gori, M., Monfardini, G., & Scarselli, F. (2005) A new model for learning in graph domains. In *Proceedings. 2005 IEEE international joint conference on neural networks*, volume 2, pp. 729–734. 265

Gouk, H., Frank, E., Pfahringer, B., & Cree, M. (2018) Regularisation of neural networks by enforcing lipschitz continuity. *arXiv preprint arXiv:1804.04368* . 154

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., & He, K., (2018) Accurate, large minibatch sgd: Training imagenet in 1 hour. 91, 240

Grathwohl, W., Chen, R. T., Bettencourt, J., Sutskever, I., & Duvenaud, D. (2018) Ffjord: Free-form continuous dynamics for scalable reversible generative models. *arXiv preprint arXiv:1810.01367* . 202

Grattarola, D., Zambon, D., Bianchi, F. M., & Alippi, C. (2022) Understanding pooling in graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems* . 268

Greydanus, S., (2020) Scaling down deep learning. 116

Griewank, A., & Walther, A. (2008) *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM. 109

Gu, J., Kwon, H., Wang, D., Ye, W., Li, M., Chen, Y.-H., Lai, L., Chandra, V., & Pan, D. Z. (2022) Multi-scale high-resolution vision transformer for semantic segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 12094–12103. 240

Ha, D., Dai, A., & Le, Q. V. (2016) Hypernetworks. *arXiv preprint arXiv:1609.09106* . 237

Hamilton, W., Ying, Z., & Leskovec, J. (2017a) Inductive representation learning on large graphs. *Advances in neural information processing systems* **30**. 265, 266, 267, 268, 270

Hamilton, W. L. (2020) Graph representation learning. *Synthesis Lectures on Artifical Intelligence and Machine Learning* **14** (3): 1–159. 265

Hamilton, W. L., Ying, R., & Leskovec, J. (2017b) Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584* . 266

Hannun, A., Case, C., Casper, J., Catanzaro, B., Diamos, G., Elsen, E., Prenger, R., Satheesh, S., Sengupta, S., Coates, A., & others. (2014) Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567* . 157

Hanson, S., & Pratt, L. (1988) Comparing biases for minimal network construction with backpropagation. *Advances in neural information processing systems* **1**. 152

Hasanzadeh, A., Hajiramezanali, E., Boluki, S., Zhou, M., Duffield, N., Narayanan, K., & Qian, X. (2020) Bayesian graph neural networks with adaptive connection sampling. In *International conference on machine learning*, pp. 4094–4104. PMLR. 268

Hayou, S., Clerico, E., He, B., Deligiannidis, G., Doucet, A., & Rousseau, J. (2021) Stable resnet. In *International Conference on Artificial Intelligence and Statistics*, pp. 1324–1332. PMLR. 205

He, K., Zhang, X., Ren, S., & Sun, J. (2015) Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR* **abs/1502.01852**. 32, 109, 182

He, K., Zhang, X., Ren, S., & Sun, J. (2016a) Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778. 187, 201

He, K., Zhang, X., Ren, S., & Sun, J. (2016b) Identity mappings in deep residual networks. In *European conference on computer vision*, pp. 630–645. Springer. 201

He, P., Liu, X., Gao, J., & Chen, W. (2020a) Deberta: Decoding-enhanced bert with disentangled attention. *arXiv preprint arXiv:2006.03654* . 239

He, X., Haffari, G., & Norouzi, M. (2020b) Dynamic programming encoding for subword segmentation in neural machine translation. *arXiv preprint arXiv:2005.06606* . 237

Henaff, M., Bruna, J., & LeCun, Y. (2015) Deep convolutional networks on graph-structured data. *arXiv preprint arXiv:1506.05163* . 265

Hendrycks, D., & Gimpel, K. (2016) Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415* . 32

Hinton, G., Srivastava, N., & Swersky, K., (2012a) Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. `https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`. 91

Hinton, G., & van Camp, D. (1993) Keeping neural networks simple by minimising the description length of weights. 1993. In *Proceedings of COLT-93*, pp. 5–13. 156

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. R. (2012b) Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580* . 156

Hochreiter, S., & Schmidhuber, J. (1997) Long short-term memory. *Neural computation* **9** (8): 1735–1780. 235

Hoffer, E., Hubara, I., & Soudry, D. (2017) Train longer, generalize better: closing the generalization gap in large batch training of neural networks. *Advances in neural information processing systems* **30**. 204

Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., Casas, D. d. L., Hendricks, L. A., Welbl, J., Clark, A., & others. (2022) Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556* . 236

Holtzman, A., Buys, J., Du, L., Forbes, M., & Choi, Y. (2019) The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751* . 237

Hornik, K. (1991) Approximation capabilities of multilayer feedforward networks. *Neural Networks* **4** (2): 251–257. 32

Howard, A., Sandler, M., Chu, G., Chen, L.-C., Chen, B., Tan, M., Wang, W., Zhu, Y., Pang, R., Vasudevan, V., & others. (2019) Searching for mobilenetv3. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 1314–1324. 32

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., & Adam, H. (2017) Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* . 181

Hu, H., Gu, J., Zhang, Z., Dai, J., & Wei, Y. (2018) Relation networks for object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3588–3597. 240

Hu, H., Zhang, Z., Xie, Z., & Lin, S. (2019) Local relation networks for image recognition. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 3464–3473. 241

Hu, W., Pang, J., Liu, X., Tian, D., Lin, C.-W., & Vetro, A. (2021) Graph signal processing for geometric data and beyond: Theory and applications. *IEEE Transactions on Multimedia* . 245

Huang, G., Li, Y., Pleiss, G., Liu, Z., Hopcroft, J. E., & Weinberger, K. Q. (2017a) Snapshot ensembles: Train 1, get m for free. *arXiv preprint arXiv:1704.00109* . 155

Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K. Q. (2017b) Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708. 205

Huang, G., Sun, Y., Liu, Z., Sedra, D., & Weinberger, K. Q. (2016) Deep networks with stochastic depth. In *European conference on computer vision*, pp. 646–661. Springer. 202

Huang, W., Zhang, T., Rong, Y., & Huang, J. (2018) Adaptive sampling towards fast graph representation learning. *Advances in neural information processing systems* **31**. 267, 268

Huang, X. S., Perez, F., Ba, J., & Volkovs, M. (2020a) Improving transformer optimization through better initialization. In *International Conference on Machine Learning*, pp. 4475–4483. PMLR. 109, 240

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, M. X., Chen, D., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., & Chen, Z., (2019) Gpipe: Efficient training of giant neural networks using pipeline parallelism. 109, 110

Huang, Z., Liang, D., Xu, P., & Xiang, B. (2020b) Improve transformer models with better relative position embeddings. *arXiv preprint arXiv:2009.13658* . 239

Huszár, F. (2019) Exponentially growing learning rate? implications of scale invariance induced by batch normalization. 204

Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2011) Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pp. 507–523. Springer. 133

Iglovikov, V., & Shvets, A. (2018) Ternausnet: U-net with vgg11 encoder pre-trained on imagenet for image segmentation. *arXiv preprint arXiv:1801.05746* . 205

Inoue, H. (2018) Data augmentation by pairing samples for images classification. *arXiv preprint arXiv:1801.02929* . 157

Ioffe, S. (2017) Batch renormalization: Towards reducing minibatch dependence in batch-normalized models. *Advances in neural information processing systems* **30**. 203

Ioffe, S., & Szegedy, C., (2015) Batch normalization: Accelerating deep network training by reducing internal covariate shift. 109, 203, 204

Ishida, T., Yamane, I., Sakai, T., Niu, G., & Sugiyama, M. (2020) Do we need zero training loss after achieving zero training error? *arXiv preprint arXiv:2002.08709* . 131, 156

Isola, P., Zhu, J.-Y., Zhou, T., & Efros, A. A. (2017) Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1125–1134. 205

Izmailov, P., Podoprikhin, D., Garipov, T., Vetrov, D., & Wilson, A. G. (2018) Averaging weights leads to wider optima and better generalization. *arXiv preprint arXiv:1803.05407* . 155

Jackson, P. T., Abarghouei, A. A., Bonner, S., Breckon, T. P., & Obara, B. (2019) Style augmentation: data augmentation via style randomization. In *CVPR Workshops*, volume 6, pp. 10–11. 157

Jacobs, R. A., Jordan, M. I., Nowlan, S. J., & Hinton, G. E. (1991) Adaptive mixtures of

local experts. *Neural computation* **3** (1): 79–87. 70

Jaitly, N., & Hinton, G. E. (2013) Vocal tract length perturbation (vtlp) improves speech recognition. In *Proc. ICML Workshop on Deep Learning for Audio, Speech and Language*, volume 117, p. 21. 157

Jarrett, K., Kavukcuoglu, K., Ranzato, M., & LeCun, Y. (2009) What is the best multi-stage architecture for object recognition? In *2009 IEEE 12th International Conference on Computer Vision*, pp. 2146–2153. 31

Ji, S., Xu, W., Yang, M., & Yu, K. (2012) 3d convolutional neural networks for human action recognition. *IEEE transactions on pattern analysis and machine intelligence* **35** (1): 221–231. 182

Jia, X., De Brabandere, B., Tuytelaars, T., & Gool, L. V. (2016) Dynamic filter networks. *Advances in neural information processing systems* **29**. 183

Jing, L., & Tian, Y. (2020) Self-supervised visual feature learning with deep neural networks: A survey. *IEEE transactions on pattern analysis and machine intelligence* **43** (11): 4037–4058. 157

Johnson, R., & Zhang, T. (2013) Accelerating stochastic gradient descent using predictive variance reduction. *Advances in neural information processing systems* **26**: 315–323. 89

Jurafsky, D., & Martin, J. H. (2000) Speech and language processing, 2nd edition. 235

Kanazawa, A., Sharma, A., & Jacobs, D. (2014) Locally scale-invariant convolutional neural networks. *arXiv preprint arXiv:1412.5104* . 182

Kanda, N., Takeda, R., & Obuchi, Y. (2013) Elastic spectral distortion for low resource speech recognition with deep neural networks. In *2013 IEEE Workshop on Automatic Speech Recognition and Understanding*, pp. 309–314. IEEE. 157

Kang, G., Dong, X., Zheng, L., & Yang, Y. (2017) Patchshuffle regularization. *arXiv preprint arXiv:1707.07103* . 157

Katharopoulos, A., Vyas, A., Pappas, N., & Fleuret, F. (2020) Transformers are rnns: Fast autoregressive transformers with linear attention. In *International Conference on Machine Learning*, pp. 5156–5165. PMLR. 240

Ke, G., He, D., & Liu, T.-Y. (2020) Rethinking positional encoding in language pre-training. *arXiv preprint arXiv:2006.15595* . 239

Kearnes, S., McCloskey, K., Berndl, M., Pande, V., & Riley, P. (2016) Molecular graph convolutions: moving beyond fingerprints. *Journal of computer-aided molecular design* **30** (8): 595–608. 266

Kendall, A., & Gal, Y. (2017) What uncertainties do we need in bayesian deep learning for computer vision? *Advances in neural information processing systems* **30**. 156

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., & Tang, P. T. P., (2017) On large-batch training for deep learning: Generalization gap and sharp minima. 156

Keskar, N. S., & Socher, R., (2017) Improving generalization performance by switching from adam to sgd. 92

Khan, S., Naseer, M., Hayat, M., Zamir, S. W., Khan, F. S., & Shah, M. (2021) Transformers in vision: A survey. *ACM Computing Surveys (CSUR)* . 240

Kingma, D. P., & Ba, J. (2014) Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* . 91, 240

Kipf, T. N., & Welling, M. (2016a) Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* . 265, 266, 267, 268

Kipf, T. N., & Welling, M. (2016b) Variational graph auto-encoders. *arXiv preprint arXiv:1611.07308* . 157

Kiranyaz, S., Avci, O., Abdeljaber, O., Ince, T., Gabbouj, M., & Inman, D. J. (2021) 1d convolutional neural networks and applications: A survey. *Mechanical systems and signal processing* **151**: 107398. 182

Kiranyaz, S., Ince, T., Hamila, R., & Gabbouj, M. (2015) Convolutional neural networks for patient-specific ecg classification. In *2015 37th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pp. 2608–2611. IEEE. 182

Kitaev, N., Kaiser, Ł., & Levskaya, A. (2020) Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451* . 239

Klambauer, G., Unterthiner, T., Mayr, A., & Hochreiter, S. (2017) Self-normalizing neural networks. In *Proceedings of the 31st international conference on neural information processing systems*, pp. 972–981. 32, 109

Koenker, R., & Hallock, K. F. (2001) Quantile regression. *Journal of economic perspectives* **15** (4): 143–156. 70

Kolomiyets, O., Bethard, S., & Moens, M.-F. (2011) Model-portability experiments for textual temporal analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: human language technologies*, volume 2, pp. 271–276. ACL; East Stroudsburg, PA. 158

Kolter, Z., Duvenaud, D., & Johnson, M. (2020) Deep implicit layers - neural odes, deep equilibirum models, and beyond. 202

Krizhevsky, A., & Hinton, G. (2009) Learning multiple layers of features from tiny images. 187

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012) Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, ed. by F. Pereira, C. J. C. Burges, L. Bottou, & K. Q. Weinberger, volume 25. Curran Associates, Inc. 49, 109, 157, 174, 180

Kudo, T. (2018) Subword regularization: Improving neural network translation models with multiple subword candidates. *arXiv preprint arXiv:1804.10959* . 237

Kudo, T., & Richardson, J. (2018) Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226* . 237

Kukačka, J., Golkov, V., & Cremers, D. (2017) Regularization for deep learning: A taxonomy. *arXiv preprint arXiv:1710.10686* . 152

Kulikov, I., Miller, A. H., Cho, K., & Weston, J. (2018) Importance of search and evaluation strategies in neural dialogue modeling. *arXiv preprint arXiv:1811.00907* . 237

Kumar, A., Singh, S. S., Singh, K., & Biswas, B. (2020) Link prediction techniques, applications, and performance: A survey. *Physica A: Statistical Mechanics and its Applications* **553**: 124289. 265

Kumar, M., Weissenborn, D., & Kalchbrenner, N. (2021) Colorization transformer. *arXiv preprint arXiv:2102.04432* . 240

Kurenkov, A. (2020) A brief history of neural nets and deep learning. *Skynet Today* . 31

Lakshminarayanan, B., Pritzel, A., & Blundell, C. (2017) Simple and scalable predictive uncertainty estimation using deep ensembles. *Advances in neural information processing systems* **30**. 155

Lample, G., & Charton, F. (2020) Deep learning for symbolic mathematics. In *International Conference on Learning Representations*. 236

Lasseck, M. (2018) Acoustic bird detection with deep convolutional neural networks. In *Proceedings of the Detection and Classification of Acoustic Scenes and Events 2018 Workshop (DCASE2018)*, pp. 143–147. 157

Lattimore, T., & Szepesvári, C. (2020) *Bandit algorithms*. Cambridge University Press. 133

Lawrence, S., Giles, C. L., Tsoi, A. C., & Back, A. D. (1997) Face recognition: A convolutional neural-network approach. *IEEE transactions on neural networks* **8** (1): 98–113. 180

Lecun, Y. (1985) Une procedure d'apprentissage pour reseau a seuil asymmetrique (a learning scheme for asymmetric threshold networks). In *Proceedings of Cognitiva 85, Paris, France*, pp. 599–604. 108

LeCun, Y., Bengio, Y., & Hinton, G. (2015) Deep learning. *nature* **521** (7553): 436–444. 49

LeCun, Y., Boser, B., Denker, J., Henderson, D., Howard, R., Hubbard, W., & Jackel, L. (1989a) Handwritten digit recognition with a back-propagation network. *Advances in neural information processing systems* **2**. 180, 182

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989b) Backpropagation applied to handwritten zip code recognition. *Neural computation* **1** (4): 541–551. 180

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998a) Gradient-based learning applied to document recognition. *Proceedings of the IEEE* **86** (11): 2278–2324. 157, 180

LeCun, Y., Bottou, L., Orr, G. B., & Müller, K. R. (1998b) *Efficient BackProp*, pp. 9–50. Springer Berlin Heidelberg. 109

Lee, J., Lee, I., & Kang, J. (2019) Self-attention graph pooling. In *International conference on machine learning*, pp. 3734–3743. PMLR. 268

Lee, J. B., Rossi, R. A., Kong, X., Kim, S., Koh, E., & Rao, A. (2018) Higher-order graph convolutional networks. *arXiv preprint arXiv:1809.07697* . 266

Li, C., Chen, C., Carlson, D., & Carin, L. (2016a) Preconditioned stochastic gradient langevin dynamics for deep neural networks. In *Thirtieth AAAI Conference on Artificial Intelligence*. 156

Li, G., Müller, M., Ghanem, B., & Koltun, V. (2021a) Training graph neural networks with

1000 layers. In *International conference on machine learning*, pp. 6437–6449. PMLR. 269

Li, G., Müller, M., Qian, G., Perez, I. C. D., Abualshour, A., Thabet, A. K., & Ghanem, B. (2021b) Deepgcns: Making gcns go as deep as cnns. *IEEE Transactions on Pattern Analysis and Machine Intelligence* . 269

Li, G., Xiong, C., Thabet, A., & Ghanem, B. (2020a) Deepergcn: All you need to train deeper gcns. *arXiv preprint arXiv:2006.07739* . 269

Li, H., Xu, Z., Taylor, G., Studer, C., & Goldstein, T. (2018a) Visualizing the loss landscape of neural nets. *Advances in neural information processing systems* **31**. 200, 201

Li, L., Jamieson, K. G., DeSalvo, G., Rostamizadeh, A., & Talwalkar, A. (2016b) Efficient hyperparameter optimization and infinitely many armed bandits. *CoRR* **abs/1603.06560**. 133

Li, L. H., Yatskar, M., Yin, D., Hsieh, C.-J., & Chang, K.-W. (2019) Visualbert: A simple and performant baseline for vision and language. *arXiv preprint arXiv:1908.03557* . 240

Li, Q., Han, Z., & Wu, X.-M. (2018b) Deeper insights into graph convolutional networks for semi-supervised learning. In *Thirty-Second AAAI conference on artificial intelligence*. 268

Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., & Chintala, S., (2020b) Pytorch distributed: Experiences on accelerating data parallel training. 110

Li, W., Lin, Z., Zhou, K., Qi, L., Wang, Y., & Jia, J. (2022) Mat: Mask-aware transformer for large hole image inpainting. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 10758–10768. 240

Li, Y., Cohn, T., & Baldwin, T. (2017) Robust training under linguistic adversity. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*, pp. 21–27. 158

Li, Y., Tarlow, D., Brockschmidt, M., & Zemel, R. (2015) Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493* . 265

Li, Z., & Arora, S. (2019) An exponential learning rate schedule for deep learning. *arXiv preprint arXiv:1910.07454* . 204

Lin, M., Chen, Q., & Yan, S. (2013) Network in network. *arXiv preprint arXiv:1312.4400* . 180, 181

Lin, T., Goyal, P., Girshick, R. B., He, K., & Dollár, P. (2017) Focal loss for dense object detection. *CoRR* **abs/1708.02002**. 70

Lin, T., Wang, Y., Liu, X., & Qiu, X. (2021) A survey of transformers. *arXiv preprint arXiv:2106.04554* . 235

Liu, G., Reda, F. A., Shih, K. J., Wang, T.-C., Tao, A., & Catanzaro, B. (2018a) Image inpainting for irregular holes using partial convolutions. In *Proceedings of the European conference on computer vision (ECCV)*, pp. 85–100. 181

Liu, L., Jiang, H., He, P., Chen, W., Liu, X., Gao, J., & Han, J., (2021a) On the variance of the adaptive learning rate and beyond. 91

Liu, L., Liu, X., Gao, J., Chen, W., & Han, J. (2020) Understanding the difficulty of training transformers. *arXiv preprint arXiv:2004.08249* . 240

Liu, L., Luo, Y., Shen, X., Sun, M., & Li, B. (2019) Beta-dropout: A unified dropout. *IEEE Access* **7**: 36140–36153. 156

Liu, P. J., Saleh, M., Pot, E., Goodrich, B., Sepassi, R., Kaiser, L., & Shazeer, N. (2018b) Generating wikipedia by summarizing long sequences. *arXiv preprint arXiv:1801.10198* . 239

Liu, X., Zhang, F., Hou, Z., Mian, L., Wang, Z., Zhang, J., & Tang, J. (2021b) Self-supervised learning: Generative or contrastive. *IEEE Transactions on Knowledge and Data Engineering* . 157

Liu, Y., Zhang, Y., Wang, Y., Hou, F., Yuan, J., Tian, J., Zhang, Y., Shi, Z., Fan, J., & He, Z. (2021c) A survey of visual transformers. *arXiv preprint arXiv:2111.06091* . 240

Liu, Z., Hu, H., Lin, Y., Yao, Z., Xie, Z., Wei, Y., Ning, J., Cao, Y., Zhang, Z., Dong, L., Wei, F., & Guo, B. (2022) Swin transformer v2: Scaling up capacity and resolution. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 12009–12019. 241

Liu, Z., Lin, Y., Cao, Y., Hu, H., Wei, Y., Zhang, Z., Lin, S., & Guo, B. (2021d) Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 10012–10022. 234, 241

Liu, Z., Sun, M., Zhou, T., Huang, G., & Darrell, T. (2018c) Rethinking the value of network pruning. *CoRR* **abs/1810.05270**. 237

Locatello, F., Weissenborn, D., Unterthiner, T., Mahendran, A., Heigold, G., Uszkoreit, J., Dosovitskiy, A., & Kipf, T. (2020) Object-centric learning with slot attention. *Advances in Neural Information Processing Systems* **33**: 11525–11538. 240

Long, J., Shelhamer, E., & Darrell, T. (2015) Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3431–3440. 181

Loshchilov, I., & Hutter, F. (2017) Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* . 92, 153

Louizos, C., Welling, M., & Kingma, D. P., (2018) Learning sparse neural networks through l_0 regularization. 153

Lu, J., Batra, D., Parikh, D., & Lee, S. (2019) Vilbert: Pretraining task-agnostic visiolinguistic representations for vision-and-language tasks. *Advances in neural information processing systems* **32**. 240

Lu, Z., Pu, H., Wang, F., Hu, Z., & Wang, L. (2017) The expressive power of neural networks: A view from the width. *CoRR* **abs/1709.02540**. 49

Lubana, E. S., Dick, R., & Tanaka, H. (2021) Beyond batchnorm: towards a unified understanding of normalization in deep learning. *Advances in Neural Information Processing Systems* **34**: 4778–4791. 203

Luo, P., Wang, X., Shao, W., & Peng, Z. (2018) Towards understanding regularization in batch normalization. *arXiv preprint arXiv:1809.00846* . 204

Luong, M.-T., Pham, H., & Manning, C. D. (2015) Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025* . 237

Luther, K. (2020) Why batch norm causes exploding gradients. 203

Ma, Y., & Tang, J. (2021) *Deep Learning on Graphs*. Cambridge University Press. 265

Ma, Y.-A., Chen, T., & Fox, E. (2015) A complete recipe for stochastic gradient mcmc. *Advances in neural information processing systems* **28**. 156

Maas, A. L., Hannun, A. Y., & Ng, A. Y. (2013) Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, p. 3. 32

MacKay, D. J., & others. (1995) Ensemble learning and evidence maximization. In *Proc. Nips*, volume 10, p. 4083. Citeseer. 156

Mahendran, A., & Vedaldi, A. (2015) Understanding deep image representations by inverting them. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 5188–5196. 183

Manning, C., & Schutze, H. (1999) *Foundations of statistical natural language processing*. MIT press. 235

Martin, G. L. (1993) Centered-object integrated segmentation and recognition of overlapping handprinted characters. *Neural Computation* **5** (3): 419–429. 180

Masci, J., Boscaini, D., Bronstein, M., & Vandergheynst, P. (2015) Geodesic convolutional neural networks on riemannian manifolds. In *Proceedings of the IEEE international conference on computer vision workshops*, pp. 37–45. 268

Maturana, D., & Scherer, S. (2015) Voxnet: A 3d convolutional neural network for real-time object recognition. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 922–928. IEEE. 182

McCoy, R. T., Pavlick, E., & Linzen, T. (2019) Right for the wrong reasons: Diagnosing syntactic heuristics in natural language inference. *arXiv preprint arXiv:1902.01007* . 236

McCulloch, W. S., & Pitts, W. (1943) A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics* **5** (4): 115–133. 31

Mezard, M., & Mora, T., (2008) Constraint satisfaction problems and neural networks: a statistical physics perspective. 90

Milletari, F., Navab, N., & Ahmadi, S.-A. (2016) V-net: Fully convolutional neural networks for volumetric medical image segmentation. In *2016 fourth international conference on 3D vision (3DV)*, pp. 565–571. IEEE. 205

Min, J., McCoy, R. T., Das, D., Pitler, E., & Linzen, T. (2020) Syntactic data augmentation increases robustness to inference heuristics. *arXiv preprint arXiv:2004.11999* . 158

Minsky, M., & Papert, S. A. (1969) *Perceptrons: An introduction to computational geometry*. MIT press. 31, 235

Mishkin, D., & Matas, J., (2016) All you need is a good init. 109

Monti, F., Boscaini, D., Masci, J., Rodola, E., Svoboda, J., & Bronstein, M. M. (2017) Geometric deep learning on graphs and manifolds using mixture model cnns. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 5115–5124. 266, 268

Monti, F., Shchur, O., Bojchevski, A., Litany, O., Günnemann, S., & Bronstein, M. M. (2018) Dual-primal graph convolutional networks. *arXiv preprint arXiv:1806.00770* . 267

Montúfar, G. (2017) Notes on the number of linear regions of deep neural networks. 49

Montúfar, G., Pascanu, R., Cho, K., & Bengio, Y. (2014) On the number of linear regions of deep neural networks. *arXiv preprint arXiv:1402.1869* . 49

Moreno-Torres, J. G., Raeder, T., Alaiz-Rodríguez, R., Chawla, N. V., & Herrera, F. (2012) A unifying view on dataset shift in classification. *Pattern recognition* **45** (1): 521–530. 132

Müller, R., Kornblith, S., & Hinton, G. E. (2019) When does label smoothing help? *Advances in neural information processing systems* **32**. 156

Mun, S., Shon, S., Kim, W., Han, D. K., & Ko, H. (2017) Deep neural network based learning and transferring mid-level audio features for acoustic scene classification. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 796–800. 157

Murphy, R. L., Srinivasan, B., Rao, V., & Ribeiro, B. (2018) Janossy pooling: Learning deep permutation-invariant functions for variable-size inputs. *arXiv preprint arXiv:1811.01900* . 266

Mutlu, E. C., Oghaz, T., Rajabi, A., & Garibay, I. (2020) Review on learning and extracting graph features for link prediction. *Machine Learning and Knowledge Extraction* **2** (4): 672–704. 265

Nair, V., & Hinton, G. E. (2010) Rectified linear units improve restricted boltzmann machines. In *Icml*. 31

Nakkiran, P., Kaplun, G., Bansal, Y., Yang, T., Barak, B., & Sutskever, I., (2019) Deep double descent: Where bigger models and more data hurt. 127, 131

Narang, S., Chung, H. W., Tay, Y., Fedus, W., Fevry, T., Matena, M., Malkan, K., Fiedel, N., Shazeer, N., Lan, Z., & others. (2021) Do transformer modifications transfer across implementations and applications? *arXiv preprint arXiv:2102.11972* . 235

Narayanan, D., Phanishayee, A., Shi, K., Chen, X., & Zaharia, M., (2021a) Memory-efficient pipeline-parallel dnn training. 110

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V. A., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., & Zaharia, M., (2021b) Efficient large-scale language model training on gpu clusters using megatron-lm. 110

Neal, R. M. (1995) *Bayesian learning for neural networks*, volume 118. Springer Science & Business Media. 156

Neimark, D., Bar, O., Zohar, M., & Asselmann, D. (2021) Video transformer network. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 3163–3172. 241

Nesterov, Y. E. (1983) A method for solving the convex programming problem with convergence rate o (1/k^ 2). In *Dokl. akad. nauk Sssr*, volume 269, pp. 543–547. 90

Newell, A., Yang, K., & Deng, J. (2016) Stacked hourglass networks for human pose estimation. In *European conference on computer vision*, pp. 483–499. Springer. 199, 205

Neyshabur, B., Bhojanapalli, S., McAllester, D., & Srebro, N., (2017a) Exploring generalization in deep learning. 131

Neyshabur, B., Bhojanapalli, S., & Srebro, N. (2017b) A pac-bayesian approach to spectrally-normalized margin bounds for neural networks. *arXiv preprint arXiv:1707.09564* . 154

Ng, N. H., Gabriel, R. A., McAuley, J., Elkan, C., & Lipton, Z. C. (2017) Predicting surgery duration with neural heteroscedastic regression. In *Machine Learning for Healthcare Conference*, pp. 100–111. PMLR. 70

Niu, F., Recht, B., Re, C., & Wright, S. J., (2011) Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. 110

Nix, D. A., & Weigend, A. S. (1994) Estimating the mean and variance of the target probability distribution. In *Proceedings of 1994 ieee international conference on neural networks (ICNN'94)*, volume 1, pp. 55–60. IEEE. 70

Noci, L., Roth, K., Bachmann, G., Nowozin, S., & Hofmann, T. (2021) Disentangling the roles of curation, data-augmentation and the prior

in the cold posterior effect. *Advances in Neural Information Processing Systems* **34**. 157

Noh, H., Hong, S., & Han, B. (2015) Learning deconvolution network for semantic segmentation. In *Proceedings of the IEEE international conference on computer vision*, pp. 1520–1528. 177, 178, 179

Noroozi, M., & Favaro, P. (2016) Unsupervised learning of visual representations by solving jigsaw puzzles. In *European conference on computer vision*, pp. 69–84. Springer. 157

Odena, A., Dumoulin, V., & Olah, C. (2016) Deconvolution and checkerboard artifacts. *Distill* **1** (10): e3. 181

Oono, K., & Suzuki, T. (2019) Graph neural networks exponentially lose expressive power for node classification. *arXiv preprint arXiv:1905.10947* . 268

Orhan, A. E., & Pitkow, X. (2017) Skip connections eliminate singularities. *arXiv preprint arXiv:1701.09175* . 201

Park, D. S., Chan, W., Zhang, Y., Chiu, C.-C., Zoph, B., Cubuk, E. D., & Le, Q. V. (2019) Specaugment: A simple data augmentation method for automatic speech recognition. *arXiv preprint arXiv:1904.08779* . 157

Park, S., & Kwak, N. (2016) Analysis on the dropout effect in convolutional neural networks. In *Asian Conference on Computer Vision*, pp. 189–204. Springer. 182

Parker, D. B. (1985) Learning-logic: Casting the cortex of the human brain in silicon. 108

Parmar, N., Vaswani, A., Uszkoreit, J., Kaiser, L., Shazeer, N., Ku, A., & Tran, D. (2018) Image transformer. In *International conference on machine learning*, pp. 4055–4064. PMLR. 241

Pascanu, R., Montufar, G., & Bengio, Y. (2014) On the number of inference regions of deep feed forward networks with piece-wise linear activations. corr. 49

Pathak, D., Krahenbuhl, P., Donahue, J., Darrell, T., & Efros, A. A. (2016) Context encoders: Feature learning by inpainting. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2536–2544. 157

Patrick, M., Campbell, D., Asano, Y., Misra, I., Metze, F., Feichtenhofer, C., Vedaldi, A., & Henriques, J. F. (2021) Keeping your eye on the ball: Trajectory attention in video transformers. *Advances in neural information processing systems* **34**: 12493–12506. 241

Pereyra, G., Tucker, G., Chorowski, J., Kaiser, ., & Hinton, G., (2017) Regularizing neural networks by penalizing confident output distributions. 156

Pham, T., Tran, T., Phung, D., & Venkatesh, S. (2017) Column networks for collective classification. In *Thirty-first AAAI conference on artificial intelligence*. 266

Phuong, M., & Hutter, M. (2022) Formal algorithms for transformers. 235

Pintea, S. L., Tömen, N., Goes, S. F., Loog, M., & van Gemert, J. C. (2021) Resolution learning in deep convolutional networks using scale-space theory. *IEEE Transactions on Image Processing* **30**: 8342–8353. 183

Poggio, T. A., Mhaskar, H., Rosasco, L., Miranda, B., & Liao, Q. (2016) Why and when can deep - but not shallow - networks avoid the curse of dimensionality: a review. *CoRR* **abs/1611.00740**. 49

Polyak, B. (1964) Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics* **4** (5): 1–17. 90

Prince, S. J. (2012) *Computer vision: models, learning, and inference*. Cambridge University Press. 156

Prince, S. J. (2021a) Transformers ii: extensions. 238, 240

Prince, S. J. (2021b) Transformers iii: training. 240

Prokudin, S., Gehler, P., & Nowozin, S., (2018) Deep directional statistics: Pose estimation with uncertainty quantification. 70

Provilkov, I., Emelianenko, D., & Voita, E. (2019) Bpe-dropout: Simple and effective subword regularization. *arXiv preprint arXiv:1910.13267* . 237

Qi, J., Du, J., Siniscalchi, S. M., Ma, X., & Lee, C.-H. (2020) On mean absolute error for deep neural network based vector-to-vector regression. *IEEE Signal Processing Letters* **27**: 1485–1489. 70

Qin, Z., Yu, F., Liu, C., & Chen, X. (2018) How convolutional neural network see the world-a survey of convolutional neural network visualization methods. *arXiv preprint arXiv:1804.11191* . 183

Qiu, S., Xu, B., Zhang, J., Wang, Y., Shen, X., De Melo, G., Long, C., & Li, X. (2020) Easyaug: An automatic textual data augmentation platform for classification tasks. In

Companion Proceedings of the Web Conference 2020, pp. 249–252. 158

Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., & others. (2021) Learning transferable visual models from natural language supervision. In *International Conference on Machine Learning*, pp. 8748–8763. PMLR. 241

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., & others. (2019) Language models are unsupervised multitask learners. *OpenAI blog* **1** (8): 9. 157, 236

Rae, J. W., Borgeaud, S., Cai, T., Millican, K., Hoffmann, J., Song, F., Aslanides, J., Henderson, S., Ring, R., Young, S., & others. (2021) Scaling language models: Methods, analysis & insights from training gopher. *arXiv preprint arXiv:2112.11446* . 236

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., Liu, P. J., & others. (2020) Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.* **21** (140): 1–67. 239

Rajpurkar, P., Zhang, J., Lopyrev, K., & Liang, P. (2016) Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250* . 236

Ramachandran, P., Parmar, N., Vaswani, A., Bello, I., Levskaya, A., & Shlens, J. (2019) Stand-alone self-attention in vision models. *Advances in Neural Information Processing Systems* **32**. 241

Ramachandran, P., Zoph, B., & Le, Q. V., (2017) Searching for activation functions. 32

Ramesh, A., Dhariwal, P., Nichol, A., Chu, C., & Chen, M. (2022) Hierarchical text-conditional image generation with clip latents. *arXiv preprint arXiv:2204.06125* . 241

Ramesh, A., Pavlov, M., Goh, G., Gray, S., Voss, C., Radford, A., Chen, M., & Sutskever, I. (2021) Zero-shot text-to-image generation. In *International Conference on Machine Learning*, pp. 8821–8831. PMLR. 241

Ramsauer, H., Schäfl, B., Lehner, J., Seidl, P., Widrich, M., Adler, T., Gruber, L., Holzleitner, M., Pavlović, M., Sandve, G. K., & others. (2020) Hopfield networks is all you need. *arXiv preprint arXiv:2008.02217* . 238

Rawat, W., & Wang, Z. (2017) Deep convolutional neural networks for image classification: A comprehensive review. *Neural computation* **29** (9): 2352–2449. 181

Reddi, S. J., Kale, S., & Kumar, S. (2018) On the convergence of adam and beyond. In *International Conference on Learning Representations*. 91

Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016) You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788. 176, 177, 178

Riad, R., Teboul, O., Grangier, D., & Zeghidour, N. (2022) Learning strides in convolutional neural networks. *arXiv preprint arXiv:2202.01653* . 183

Ribeiro, M. T., Wu, T., Guestrin, C., & Singh, S. (2020) Beyond accuracy: Behavioral testing of nlp models with checklist. *arXiv preprint arXiv:2005.04118* . 236

Rives, A., Meier, J., Sercu, T., Goyal, S., Lin, Z., Liu, J., Guo, D., Ott, M., Zitnick, C. L., Ma, J., & others. (2021) Biological structure and function emerge from scaling unsupervised learning to 250 million protein sequences. *Proceedings of the National Academy of Sciences* **118** (15). 235

Robbins, H., & Monro, S. (1951) A stochastic approximation method. *The Annals of Mathematical Statistics* **22** (3): 400–407. 89

Rodrigues, F., & Pereira, F. C., (2018) Beyond expectation: Deep joint mean and quantile regression for spatio-temporal problems. 70

Romero, D. W., Bruintjes, R.-J., Tomczak, J. M., Bekkers, E. J., Hoogendoorn, M., & van Gemert, J. C. (2021) Flexconv: Continuous kernel convolutions with differentiable kernel sizes. *arXiv preprint arXiv:2110.08059* . 183

Rong, Y., Huang, W., Xu, T., & Huang, J. (2020) Dropedge: Towards deep graph convolutional networks on node classification. In *International Conference on Learning Representations*. 267

Ronneberger, O., Fischer, P., & Brox, T. (2015) U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pp. 234–241. Springer. 205

Rosenblatt, F. (1958) The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review* **65** (6): 386. 31

Rossi, E., Frasca, F., Chamberlain, B., Eynard, D., Bronstein, M., & Monti, F. (2020) Sign: Scalable inception graph neural networks. *arXiv preprint arXiv:2004.11198* **7**: 15. 266

Roy, A., Saffar, M., Vaswani, A., & Grangier, D. (2021) Efficient content-based sparse attention with routing transformers. *Transactions of the Association for Computational Linguistics* **9**: 53–68. 239

Rozemberczki, B., Kiss, O., & Sarkar, R. (2020) Little ball of fur: a python library for graph sampling. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, pp. 3133–3140. 267

Ruder, S. (2016) An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747* . 89

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1985) Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science. 108, 235

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986) Learning representations by back-propagating errors. *nature* **323** (6088): 533–536. 108

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., & others. (2015) Imagenet large scale visual recognition challenge. *International journal of computer vision* **115** (3): 211–252. 174, 181

Sabour, S., Frosst, N., & Hinton, G. E. (2017) Dynamic routing between capsules. *Advances in neural information processing systems* **30**. 238

Safran, I., & Shamir, O. (2016) Depth separation in relu networks for approximating smooth non-linear functions. *CoRR* **abs/1610.09887**. 49

Saha, S., Singh, G., Sapienza, M., Torr, P. H., & Cuzzolin, F. (2016) Deep learning for detecting multiple space-time action tubes in videos. *arXiv preprint arXiv:1608.01529* . 182

Sainath, T. N., Kingsbury, B., Mohamed, A.-r., Dahl, G. E., Saon, G., Soltau, H., Beran, T., Aravkin, A. Y., & Ramabhadran, B. (2013) Improvements to deep convolutional neural networks for lvcsr. In *2013 IEEE workshop on automatic speech recognition and understanding*, pp. 315–320. IEEE. 182

Salamon, J., & Bello, J. P. (2017) Deep convolutional neural networks and data augmentation for environmental sound classification. *IEEE Signal processing letters* **24** (3): 279–283. 157

Salimans, T., & Kingma, D. P. (2016) Weight normalization: A simple reparameterization to accelerate training of deep neural networks. *Advances in neural information processing systems* **29**. 203

Sanchez-Lengeling, B., Reif, E., Pearce, A., & Wiltschko, A. B. (2021) A gentle introduction to graph neural networks. *Distill* . 264

Sankararaman, K. A., De, S., Xu, Z., Huang, W. R., & Goldstein, T. (2020) The impact of neural network overparameterization on gradient confusion and stochastic gradient descent. In *International conference on machine learning*, pp. 8469–8479. PMLR. 201

Santurkar, S., Tsipras, D., Ilyas, A., & Madry, A. (2018) How does batch normalization help optimization? *Advances in neural information processing systems* **31**. 204

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., & Monfardini, G. (2008) The graph neural network model. *IEEE transactions on neural networks* **20** (1): 61–80. 265

Scherer, D., Müller, A., & Behnke, S. (2010) Evaluation of pooling operations in convolutional architectures for object recognition. In *International conference on artificial neural networks*, pp. 92–101. Springer. 182

Schlag, I., Irie, K., & Schmidhuber, J. (2021) Linear transformers are secretly fast weight programmers. In *International Conference on Machine Learning*, pp. 9355–9366. PMLR. 238

Schlichtkrull, M., Kipf, T. N., Bloem, P., Berg, R. v. d., Titov, I., & Welling, M. (2018) Modeling relational data with graph convolutional networks. In *European semantic web conference*, pp. 593–607. Springer. 267, 268

Schneider, S., Baevski, A., Collobert, R., & Auli, M. (2019) wav2vec: Unsupervised pre-training for speech recognition. *arXiv preprint arXiv:1904.05862* . 157

Schuster, M., & Nakajima, K. (2012) Japanese and korean voice search. In *2012 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pp. 5149–5152. IEEE. 237

Schwarz, J., Jayakumar, S., Pascanu, R., Latham, P., & Teh, Y. (2021) Powerpropagation: A sparsity inducing weight reparam-

eterisation. *Advances in Neural Information Processing Systems* **34**. 154

Sejnowski, T. J. (2018) *The deep learning revolution*. MIT press. 31

Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., & Dill, D. L. (2018) Learning a sat solver from single-bit supervision. *arXiv preprint arXiv:1802.03685* . 265

Selva, J., Johansen, A. S., Escalera, S., Nasrollahi, K., Moeslund, T. B., & Clapés, A. (2022) Video transformers: A survey. *arXiv preprint arXiv:2201.05991* . 241

Sennrich, R., Haddow, B., & Birch, A. (2015) Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909* . 237

Serra, T., Tjandraatmadja, C., & Ramalingam, S. (2018) Bounding and counting linear regions of deep neural networks. In *International Conference on Machine Learning*, pp. 4558–4566. PMLR. 49

Shang, W., Sohn, K., Almeida, D., & Lee, H. (2016) Understanding and improving convolutional neural networks via concatenated rectified linear units. In *international conference on machine learning*, pp. 2217–2225. 32

Sharif Razavian, A., Azizpour, H., Sullivan, J., & Carlsson, S. (2014) Cnn features off-the-shelf: an astounding baseline for recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pp. 806–813. 157

Shaw, P., Uszkoreit, J., & Vaswani, A. (2018) Self-attention with relative position representations. *arXiv preprint arXiv:1803.02155* . 239

Shen, S., Yao, Z., Gholami, A., Mahoney, M., & Keutzer, K. (2020) Powernorm: Rethinking batch normalization in transformers. In *International Conference on Machine Learning*, pp. 8741–8751. PMLR. 240

Shen, X., Tian, X., Liu, T., Xu, F., & Tao, D. (2017) Continuous dropout. *IEEE transactions on neural networks and learning systems* **29** (9): 3926–3937. 156

Shi, W., Caballero, J., Huszár, F., Totz, J., Aitken, A. P., Bishop, R., Rueckert, D., & Wang, Z. (2016) Real-time single image and video super-resolution using an efficient subpixel convolutional neural network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1874–1883. 182

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., & Catanzaro, B., (2020) Megatron-lm: Training multi-billion parameter language models using model parallelism. 110

Shorten, C., & Khoshgoftaar, T. M. (2019) A survey on image data augmentation for deep learning. *Journal of big data* **6** (1): 1–48. 157

Siddique, N., Paheding, S., Elkin, C. P., & Devabhaktuni, V. (2021) U-net and its variants for medical image segmentation: A review of theory and applications. *IEEE Access* . 205

Sifre, L., & Mallat, S. (2013) Rotation, scaling and deformation invariant scattering for texture discrimination. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1233–1240. 182

Simonyan, K., & Zisserman, A. (2014) Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* . 175, 180

Sjöberg, J., & Ljung, L. (1995) Overtraining, regularization and searching for a minimum, with application to neural networks. *International Journal of Control* **62** (6): 1391–1407. 155

Smith, S., Elsen, E., & De, S. (2020) On the generalization benefit of noise in stochastic gradient descent. In *International Conference on Machine Learning*, pp. 9058–9067. PMLR. 155

Smith, S., Patwary, M., Norick, B., LeGresley, P., Rajbhandari, S., Casper, J., Liu, Z., Prabhumoye, S., Zerveas, G., Korthikanti, V., & others. (2022) Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990* . 236

Smith, S. L., Dherin, B., Barrett, D. G. T., & De, S., (2021) On the origin of implicit regularization in stochastic gradient descent. 155

Snoek, J., Larochelle, H., & Adams, R. P. (2012) Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pp. 2951–2959. 133

Sohoni, N. S., Aberger, C. R., Leszczynski, M., Zhang, J., & Ré, C., (2019) Low-memory neural network training: A technical report. 109

Springenberg, J. T., Dosovitskiy, A., Brox, T., & Riedmiller, M. (2014) Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806* . 182

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014) Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research* **15** (1): 1929–1958. 156

Srivastava, R. K., Greff, K., & Schmidhuber, J. (2015) Highway networks. *arXiv preprint arXiv:1505.00387* . 201

Su, H., Jampani, V., Sun, D., Gallo, O., Learned-Miller, E., & Kautz, J. (2019a) Pixel-adaptive convolutional neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 11166–11175. 183

Su, J., Lu, Y., Pan, S., Wen, B., & Liu, Y. (2021) Roformer: Enhanced transformer with rotary position embedding. *arXiv preprint arXiv:2104.09864* . 239

Su, W., Zhu, X., Cao, Y., Li, B., Lu, L., Wei, F., & Dai, J. (2019b) Vl-bert: Pre-training of generic visual-linguistic representations. *arXiv preprint arXiv:1908.08530* . 240

Summers, C., & Dinneen, M. J. (2019) Improved mixed-example data augmentation. In *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pp. 1262–1270. IEEE. 157

Sun, C., Myers, A., Vondrick, C., Murphy, K., & Schmid, C. (2019) Videobert: A joint model for video and language representation learning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 7464–7473. 240

Sun, C., Shrivastava, A., Singh, S., & Gupta, A. (2017) Revisiting unreasonable effectiveness of data in deep learning era. In *Proceedings of the IEEE international conference on computer vision*, pp. 843–852. 240

Sun, R.-Y. (2020) Optimization for deep learning: An overview. *Journal of the Operations Research Society of China* **8** (2): 249–294. 89

Sutskever, I., Martens, J., Dahl, G., & Hinton, G. (2013) On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning*, ed. by S. Dasgupta & D. McAllester, volume 28 of *Proceedings of Machine Learning Research*, pp. 1139–1147. PMLR. 90

Szegedy, C., Ioffe, S., Vanhoucke, V., & Alemi, A. A. (2017) Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-first AAAI conference on artificial intelligence*. 180, 183

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016) Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2818–2826. 152, 156

Tan, H., & Bansal, M. (2019) Lxmert: Learning cross-modality encoder representations from transformers. *arXiv preprint arXiv:1908.07490* . 240

Tay, Y., Bahri, D., Metzler, D., Juan, D.-C., Zhao, Z., & Zheng, C. (2021) Synthesizer: Rethinking self-attention for transformer models. In *International conference on machine learning*, pp. 10183–10192. PMLR. 238

Tay, Y., Bahri, D., Yang, L., Metzler, D., & Juan, D.-C. (2020a) Sparse sinkhorn attention. In *International Conference on Machine Learning*, pp. 9438–9447. PMLR. 239

Tay, Y., Dehghani, M., Bahri, D., & Metzler, D. (2020b) Efficient transformers: A survey. *ACM Computing Surveys (CSUR)* . 240

Telgarsky, M. (2016) Benefits of depth in neural networks. *CoRR* . 49

Teru, K., Denis, E., & Hamilton, W. (2020) Inductive relation prediction by subgraph reasoning. In *International Conference on Machine Learning*, pp. 9448–9457. PMLR. 267

Teye, M., Azizpour, H., & Smith, K. (2018) Bayesian uncertainty estimation for batch normalized deep networks. In *International Conference on Machine Learning*, pp. 4907–4916. PMLR. 203

Thoppilan, R., De Freitas, D., Hall, J., Shazeer, N., Kulshreshtha, A., Cheng, H.-T., Jin, A., Bos, T., Baker, L., Du, Y., & others. (2022) Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239* . 236

Tompson, J., Goroshin, R., Jain, A., LeCun, Y., & Bregler, C. (2015) Efficient object localization using convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 648–656. 183

Touvron, H., Cord, M., Douze, M., Massa, F., Sablayrolles, A., & Jégou, H. (2021) Training data-efficient image transformers & distillation through attention. In *International Conference on Machine Learning*, pp. 10347–10357. PMLR. 240

Tran, D., Bourdev, L., Fergus, R., Torresani, L., & Paluri, M. (2015) Learning spatiotemporal features with 3d convolutional networks. In *Proceedings of the IEEE international conference on computer vision*, pp. 4489–4497. 182

Tran, D., Wang, H., Torresani, L., Ray, J., LeCun, Y., & Paluri, M. (2018) A closer look at spatiotemporal convolutions for action recognition. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pp. 6450–6459. 181

Tsitsulin, A., Palowitch, J., Perozzi, B., & Müller, E. (2020) Graph clustering with graph neural networks. *arXiv preprint arXiv:2006.16904* . 265

Ulyanov, D., Vedaldi, A., & Lempitsky, V. (2016) Instance normalization: The missing ingredient for fast stylization. *arXiv preprint arXiv:1607.08022* . 203

Van Oord, A., Kalchbrenner, N., & Kavukcuoglu, K. (2016) Pixel recurrent neural networks. In *International conference on machine learning*, pp. 1747–1756. PMLR. 235

Vapnik, V. (1995) *The Nature of Statistical Learning Theory*. Springer Verlag. 71

Vapnik, V. N., & Chervonenkis, A. Y. (1971) On the uniform convergence of relative frequencies of events to their probabilities. In *Measures of complexity*, pp. 11–30. Springer. 131

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017) Attention is all you need. *Advances in neural information processing systems* **30**. 156, 235, 236, 237, 238, 240

Veit, A., Wilber, M. J., & Belongie, S. (2016) Residual networks behave like ensembles of relatively shallow networks. *Advances in neural information processing systems* **29**. 202

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., & Bengio, Y. (2017) Graph attention networks. *arXiv preprint arXiv:1710.10903* . 235, 266, 267

Vijayakumar, A. K., Cogswell, M., Selvaraju, R. R., Sun, Q., Lee, S., Crandall, D., & Batra, D. (2016) Diverse beam search: Decoding diverse solutions from neural sequence models. *arXiv preprint arXiv:1610.02424* . 237

Voita, E., Talbot, D., Moiseev, F., Sennrich, R., & Titov, I. (2019) Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned. *arXiv preprint arXiv:1905.09418* . 237

Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., & Lang, K. J. (1989) Phoneme recognition using time-delay neural networks. *IEEE transactions on acoustics, speech, and signal processing* **37** (3): 328–339. 180

Wan, L., Zeiler, M., Zhang, S., Le Cun, Y., & Fergus, R. (2013) Regularization of neural networks using dropconnect. In *International conference on machine learning*, pp. 1058–1066. PMLR. 156

Wan, Z., Zhang, J., Chen, D., & Liao, J. (2021) High-fidelity pluralistic image completion with transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 4692–4701. 240

Wang, A., Pruksachatkun, Y., Nangia, N., Singh, A., Michael, J., Hill, F., Levy, O., & Bowman, S. (2019) Superglue: A stickier benchmark for general-purpose language understanding systems. *Advances in neural information processing systems* **32**. 236

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., & Bowman, S. R. (2018a) Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461* . 236

Wang, B., Shang, L., Lioma, C., Jiang, X., Yang, H., Liu, Q., & Simonsen, J. G. (2020a) On position embeddings in bert. In *International Conference on Learning Representations*. 238, 239

Wang, S., Li, B. Z., Khabsa, M., Fang, H., & Ma, H. (2020b) Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768* . 239

Wang, W., Xie, E., Li, X., Fan, D.-P., Song, K., Liang, D., Lu, T., Luo, P., & Shao, L. (2021a) Pyramid vision transformer: A versatile backbone for dense prediction without convolutions. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 568–578. 241

Wang, W., Yao, L., Chen, L., Lin, B., Cai, D., He, X., & Liu, W. (2021b) Crossformer: A versatile vision transformer hinging on cross-scale attention. *arXiv preprint arXiv:2108.00154* . 241

Wang, X., Girshick, R., Gupta, A., & He, K. (2018b) Non-local neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 7794–7803. 240

Wang, Y., Mohamed, A., Le, D., Liu, C., Xiao, A., Mahadeokar, J., Huang, H., Tjandra,

A., Zhang, X., Zhang, F., & others. (2020c) Transformer-based acoustic modeling for hybrid speech recognition. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 6874–6878. IEEE. 236

Wei, J., Ren, X., Li, X., Huang, W., Liao, Y., Wang, Y., Lin, J., Jiang, X., Chen, X., & Liu, Q. (2019) Nezha: Neural contextualized representation for chinese language understanding. *arXiv preprint arXiv:1909.00204* . 239

Wei, J., & Zou, K. (2019) Eda: Easy data augmentation techniques for boosting performance on text classification tasks. *arXiv preprint arXiv:1901.11196* . 157

Weisfeiler, B., & Leman, A. (1968) The reduction of a graph to canonical form and the algebra which appears therein. *NTI, Series* **2** (9): 12–16. 267

Welling, M., & Teh, Y. W. (2011) Bayesian learning via stochastic gradient langevin dynamics. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pp. 681–688. Citeseer. 156

Wenzel, F., Roth, K., Veeling, B. S., Świątkowski, J., Tran, L., Mandt, S., Snoek, J., Salimans, T., Jenatton, R., & Nowozin, S. (2020a) How good is the bayes posterior in deep neural networks really? *arXiv preprint arXiv:2002.02405* . 157

Wenzel, F., Snoek, J., Tran, D., & Jenatton, R. (2020b) Hyperparameter ensembles for robustness and uncertainty quantification. *Advances in Neural Information Processing Systems* **33**: 6514–6527. 155

Werbos, P. (1974) Beyond regression:" new tools for prediction and analysis in the behavioral sciences. *Ph. D. dissertation, Harvard University* . 108

Whitney, H. (1932) Congruent graphs and the connectivity of graphs. In *Hassler Whitney Collected Papers*, pp. 61–79. Springer. 266

Wightman, R., Touvron, H., & Jégou, H. (2021) Resnet strikes back: An improved training procedure in timm. *arXiv preprint arXiv:2110.00476* . 201

Williams, P. M. (1996) Using neural networks to model conditional multivariate densities. *Neural computation* **8** (4): 843–854. 70

Wilson, A. C., Roelofs, R., Stern, M., Srebro, N., & Recht, B., (2018) The marginal value of adaptive gradient methods in machine learning. 92

Wolfe, C. R., Yang, J., Chowdhury, A., Dun, C., Bayer, A., Segarra, S., & Kyrillidis, A. (2021) Gist: Distributed training for large-scale graph convolutional networks. *arXiv preprint arXiv:2102.10424* . 267

Wolpert, D. H. (1992) Stacked generalization. *Neural networks* **5** (2): 241–259. 155

Worrall, D. E., Garbin, S. J., Turmukhambetov, D., & Brostow, G. J. (2017) Harmonic networks: Deep translation and rotation equivariance. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5028–5037. 182

Wu, B., Xu, C., Dai, X., Wan, A., Zhang, P., Yan, Z., Tomizuka, M., Gonzalez, J., Keutzer, K., & Vajda, P. (2020a) Visual transformers: Token-based image representation and processing for computer vision. *arXiv preprint arXiv:2006.03677* . 240

Wu, F., Fan, A., Baevski, A., Dauphin, Y. N., & Auli, M. (2019) Pay less attention with lightweight and dynamic convolutions. *arXiv preprint arXiv:1901.10430* . 238

Wu, H., & Gu, X. (2015) Max-pooling dropout for regularization of convolutional neural networks. In *International Conference on Neural Information Processing*, pp. 46–54. Springer. 183

Wu, N., Green, B., Ben, X., & O'Banion, S. (2020b) Deep transformer models for time series forecasting: The influenza prevalence case. *arXiv preprint arXiv:2001.08317* . 236

Wu, R., Yan, S., Shan, Y., Dang, Q., & Sun, G., (2015a) Deep image: Scaling up image recognition. 151

Wu, S., Sun, F., Zhang, W., Xie, X., & Cui, B. (2020c) Graph neural networks in recommender systems: a survey. *ACM Computing Surveys (CSUR)* . 265

Wu, Y., & He, K. (2018) Group normalization. In *Proceedings of the European conference on computer vision (ECCV)*, pp. 3–19. 203, 204

Wu, Z., Nagarajan, T., Kumar, A., Rennie, S., Davis, L. S., Grauman, K., & Feris, R. (2018) Blockdrop: Dynamic inference paths in residual networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 8817–8826. 202

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., & Philip, S. Y. (2020d) A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* **32** (1): 4–24. 264

Wu, Z., Song, S., Khosla, A., Yu, F., Zhang, L., Tang, X., & Xiao, J. (2015b) 3d shapenets: A deep representation for volumetric shapes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1912–1920. 182

Xia, F., Liu, T.-Y., Wang, J., Zhang, W., & Li, H. (2008) Listwise approach to learning to rank: theory and algorithm. In *Proceedings of the 25th international conference on Machine learning*, pp. 1192–1199. 70

Xiao, L., Bahri, Y., Sohl-Dickstein, J., Schoenholz, S., & Pennington, J. (2018) Dynamical isometry and a mean field theory of cnns: How to train 10,000-layer vanilla convolutional neural networks. In *International Conference on Machine Learning*, pp. 5393–5402. PMLR. 109, 182

Xiao, S., Wang, S., Dai, Y., & Guo, W. (2022) Graph neural networks in node classification: survey and evaluation. *Machine Vision and Applications* **33** (1): 1–19. 265

Xie, E., Wang, W., Yu, Z., Anandkumar, A., Alvarez, J. M., & Luo, P. (2021) Segformer: Simple and efficient design for semantic segmentation with transformers. *Advances in Neural Information Processing Systems* **34**: 12077–12090. 240

Xie, L., Wang, J., Wei, Z., Wang, M., & Tian, Q. (2016) Disturblabel: Regularizing cnn on the loss layer. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4753–4762. 156

Xie, S., Girshick, R., Dollár, P., Tu, Z., & He, K. (2017) Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1492–1500. 181, 202

Xing, E. P., Ho, Q., Dai, W., Kim, J. K., Wei, J., Lee, S., Zheng, X., Xie, P., Kumar, A., & Yu, Y. (2015) Petuum: A new platform for distributed machine learning on big data. *IEEE transactions on Big Data* **1** (2): 49–67. 110

Xiong, R., Yang, Y., He, D., Zheng, K., Zheng, S., Xing, C., Zhang, H., Lan, Y., Wang, L., & Liu, T. (2020a) On layer normalization in the transformer architecture. In *International Conference on Machine Learning*, pp. 10524–10533. PMLR. 240

Xiong, Z., Yuan, Y., Guo, N., & Wang, Q. (2020b) Variational context-deformable convnets for indoor scene parsing. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 3992–4002. 183

Xu, B., Wang, N., Chen, T., & Li, M. (2015) Empirical evaluation of rectified activations in convolutional network. *CoRR* **abs/1505.00853**. 156, 157

Xu, K., Hu, W., Leskovec, J., & Jegelka, S. (2019) How powerful are graph neural networks? In *International Conference on Learning Representations*. 267

Xu, K., Li, C., Tian, Y., Sonobe, T., Kawarabayashi, K.-i., & Jegelka, S. (2018) Representation learning on graphs with jumping knowledge networks. In *International conference on machine learning*, pp. 5453–5462. PMLR. 266, 268, 269

Xu, K., Zhang, M., Jegelka, S., & Kawaguchi, K. (2021a) Optimization of graph neural networks: Implicit acceleration by skip connections and more depth. In *International Conference on Machine Learning*, pp. 11592–11602. PMLR. 269

Xu, P., Kumar, D., Yang, W., Zi, W., Tang, K., Huang, C., Cheung, J. C. K., Prince, S. J., & Cao, Y. (2021b) Optimizing deeper transformers on small datasets. *arXiv preprint arXiv:2012.15355* . 109, 236, 240

Yamada, Y., Iwamura, M., Akiba, T., & Kise, K. (2019) Shakedrop regularization for deep residual learning. *IEEE Access* **7**: 186126–186136. 202

Yamada, Y., Iwamura, M., & Kise, K. (2016) Deep pyramidal residual networks with separated stochastic depth. *arXiv preprint arXiv:1612.01230* . 202

Yang, F., Yang, H., Fu, J., Lu, H., & Guo, B. (2020a) Learning texture transformer network for image super-resolution. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 5791–5800. 240

Yang, G., Pennington, J., Rao, V., Sohl-Dickstein, J., & Schoenholz, S. S. (2019) A mean field theory of batch normalization. *arXiv preprint arXiv:1902.08129* . 203

Yang, Q., Zhang, Y., Dai, W., & Pan, S. J. (2020b) *Transfer Learning*. Cambridge University Press. 157

Yao, W., Zeng, Z., Lian, C., & Tang, H. (2018) Pixel-wise regression using u-net and its application on pansharpening. *Neurocomputing* **312**: 364–371. 205

Ye, H., & Young, S. (2004) High quality voice morphing. In *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 1, pp. I–9. IEEE. 157

Ye, L., Rochan, M., Liu, Z., & Wang, Y. (2019) Cross-modal self-attention network for referring image segmentation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 10502–10511. 240

Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W. L., & Leskovec, J. (2018a) Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 974–983. 267, 268

Ying, Z., You, J., Morris, C., Ren, X., Hamilton, W., & Leskovec, J. (2018b) Hierarchical graph representation learning with differentiable pooling. *Advances in neural information processing systems* **31**. 268

Yoshida, Y., & Miyato, T. (2017) Spectral norm regularization for improving the generalizability of deep learning. *arXiv preprint arXiv:1705.10941* . 154

You, Y., Chen, T., Wang, Z., & Shen, Y. (2020) When does self-supervision help graph convolutional networks? In *international conference on machine learning*, pp. 10871–10880. PMLR. 157

Yu, F., & Koltun, V. (2015) Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122* . 181

Yu, J., Lin, Z., Yang, J., Shen, X., Lu, X., & Huang, T. S. (2019) Free-form image inpainting with gated convolution. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 4471–4480. 182

Yun, S., Han, D., Oh, S. J., Chun, S., Choe, J., & Yoo, Y. (2019) Cutmix: Regularization strategy to train strong classifiers with localizable features. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 6023–6032. 157

Zagoruyko, S., & Komodakis, N. (2016) Wide residual networks. *arXiv preprint arXiv:1605.07146* . 201

Zaheer, M., Kottur, S., Ravanbakhsh, S., Poczos, B., Salakhutdinov, R. R., & Smola, A. J. (2017) Deep sets. *Advances in neural information processing systems* **30**. 266

Zaheer, M., Reddi, S., Sachan, D., Kale, S., & Kumar, S. (2018) Adaptive methods for nonconvex optimization. *Advances in neural information processing systems* **31**. 91

Zaslavsky, T. (1975) *Facing up to arrangements: Face-count formulas for partitions of space by hyperplanes: Face-count formulas for partitions of space by hyperplanes*, volume 154. American Mathematical Soc. 32, 35

Zeiler, M. D. (2012) Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701* . 91

Zeiler, M. D., & Fergus, R. (2014) Visualizing and understanding convolutional networks. In *European conference on computer vision*, pp. 818–833. Springer. 180, 182, 183

Zeiler, M. D., Taylor, G. W., & Fergus, R. (2011) Adaptive deconvolutional networks for mid and high level feature learning. In *2011 international conference on computer vision*, pp. 2018–2025. IEEE. 182

Zeng, H., Zhou, H., Srivastava, A., Kannan, R., & Prasanna, V. (2019a) Graphsaint: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931* . 267

Zeng, Y., Fu, J., Chao, H., & Guo, B. (2019b) Learning pyramid-context encoder network for high-quality image inpainting. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1486–1494. 205

Zhai, S., Talbott, W., Srivastava, N., Huang, C., Goh, H., Zhang, R., & Susskind, J., (2021) An attention free transformer. 237

Zhang, C., Bengio, S., Hardt, M., Recht, B., & Vinyals, O. (2016a) Understanding deep learning requires rethinking generalization. *CoRR* **abs/1611.03530**. 154

Zhang, H., Cisse, M., Dauphin, Y. N., & Lopez-Paz, D. (2017) mixup: Beyond empirical risk minimization. *arXiv preprint arXiv:1710.09412* . 157

Zhang, H., Dauphin, Y. N., & Ma, T., (2019a) Fixup initialization: Residual learning without normalization. 109, 205

Zhang, H., Hsieh, C.-J., & Akella, V. (2016b) Hogwild++: A new mechanism for decentralized asynchronous stochastic gradient descent. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pp. 629–638. 110

Zhang, J., & Meng, L. (2019) Gresnet: Graph residual network for reviving deep gnns from suspended animation. *arXiv preprint arXiv:1909.05729* . 266

Zhang, J., Shi, X., Xie, J., Ma, H., King, I., & Yeung, D.-Y. (2018a) Gaan: Gated attention networks for learning on large and spatiotemporal graphs. *arXiv preprint arXiv:1803.07294* . 266

Zhang, J., Zhang, H., Xia, C., & Sun, L. (2020) Graph-bert: Only attention is needed for learning graph representations. *arXiv preprint arXiv:2001.05140* . 266

Zhang, M., & Chen, Y. (2018) Link prediction based on graph neural networks. *Advances in neural information processing systems* **31**. 265

Zhang, M., Cui, Z., Neumann, M., & Chen, Y. (2018b) An end-to-end deep learning architecture for graph classification. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32. 265, 268

Zhang, R. (2019) Making convolutional networks shift-invariant again. In *International conference on machine learning*, pp. 7324–7334. PMLR. 182

Zhang, R., Isola, P., & Efros, A. A. (2016c) Colorful image colorization. In *European conference on computer vision*, pp. 649–666. Springer. 157

Zhang, S., Tong, H., Xu, J., & Maciejewski, R. (2019b) Graph convolutional networks: a comprehensive review. *Computational Social Networks* **6** (1): 1–23. 265

Zhang, X., Zhao, J., & LeCun, Y. (2015) Character-level convolutional networks for text classification. *Advances in neural information processing systems* **28**. 182

Zhao, H., Jia, J., & Koltun, V. (2020a) Exploring self-attention for image recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 10076–10085. 241

Zhao, L., & Akoglu, L. (2019) Pairnorm: Tackling oversmoothing in gnns. *arXiv preprint arXiv:1909.12223* . 268

Zhao, L., Mo, Q., Lin, S., Wang, Z., Zuo, Z., Chen, H., Xing, W., & Lu, D. (2020b) Uctgan: Diverse image inpainting based on unsupervised cross-space translation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 5741–5750. 240

Zheng, C., Cham, T.-J., & Cai, J. (2021) Tfill: Image completion via a transformer-based architecture. *arXiv preprint arXiv:2104.00845* . 240

Zhong, Z., Zheng, L., Kang, G., Li, S., & Yang, Y. (2020) Random erasing data augmentation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pp. 13001–13008. 157

Zhou, J., Cui, G., Hu, S., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C., & Sun, M. (2020a) Graph neural networks: A review of methods and applications. *AI Open* **1**: 57–81. 264

Zhou, K., Huang, X., Li, Y., Zha, D., Chen, R., & Hu, X. (2020b) Towards deeper graph neural networks with differentiable group normalization. *Advances in neural information processing systems* **33**: 4917–4928. 268

Zhou, Y.-T., & Chellappa, R. (1988) Computation of optical flow using a neural network. In *ICNN*, pp. 71–78. 182

Zhou, Z., & Li, X. (2017) Graph convolution: a high-order and adaptive approach. *arXiv preprint arXiv:1706.09916* . 266

Zhou, Z., Rahman Siddiquee, M. M., Tajbakhsh, N., & Liang, J. (2018) Unet++: A nested u-net architecture for medical image segmentation. In *Deep learning in medical image analysis and multimodal learning for clinical decision support*, pp. 3–11. Springer. 205

Zhu, C., Ni, R., Xu, Z., Kong, K., Huang, W. R., & Goldstein, T. (2021) Gradinit: Learning to initialize neural networks for stable and efficient training. *Advances in Neural Information Processing Systems* **34**. 109

Zhu, X., Su, W., Lu, L., Li, B., Wang, X., & Dai, J. (2020) Deformable detr: Deformable transformers for end-to-end object detection. *arXiv preprint arXiv:2010.04159* . 240

Zhuang, F., Qi, Z., Duan, K., Xi, D., Zhu, Y., Zhu, H., Xiong, H., & He, Q. (2020) A comprehensive survey on transfer learning. *Proceedings of the IEEE* **109** (1): 43–76. 157

Zou, D., Hu, Z., Wang, Y., Jiang, S., Sun, Y., & Gu, Q. (2019) Layer-dependent importance sampling for training deep and large graph convolutional networks. *Advances in neural information processing systems* **32**. 267

Index